Tomography mapped onto the Cell Broadband Processor

A Master's thesis

Computer Science

Sander van der Maar
Supervisor: Dr. Ir. Bart Kienhuis
Second Reader: Drs. Sjoerd Meijer

August 9, 2007

Abstract

This document covers a project running from February to August 2007, during which we implemented a tomography algorithm (SART) in a few different ways and also set up a compiler that generated parallelized code for the IBM Cell. Here we give the background for this project and explain the used methods and algorithms. Also the found results are discussed.

Acknowledgment

This project would not have been possible without the people who helped me. I received the most support from drs. Sjoerd Meijer. From the very first idea to the final phase, it wouldn't have gone as well without his help. He already started some research on the STI Cell, which made it possible for me to have a program running during the first couple of weeks. Also his hands-on experience with the used software (COMPAAN, KPNFormat, Panda, the Cell Simulator, etc.) allowed me to pick this all up very rapidly.

A second person who was very important to this project is Dr. Ir. Bart Kienhuis. Managing the resource allocation and introducing me to a number of very experienced people turned out to be invaluable and allowed the project to get its broad scope that it eventually got. His years of experience with writing code for parallel systems and other fields inside Computer Science and outside all contributed. His eye for good software development is the reason why we have been able to set up a working back-end for COMPAAN as quickly as we did.

A large part of this project is based on previous work done by Dr. Joost Batenburg. He was kind enough to explain the inner workings of SART to me and also answered any other questions I had about tomography. His extremely wide field of interest offered the link between mathematics and computer science needed to implement a tomography algorithm.

The rest of the Leiden Embedded Research Center team at LIACS was always willing to answer questions regarding the software already available for solving parallel computation problems. Here Dmitry Nadezhkin and Bin Jiang not only helped with problems I ran into during the course of the project but also offered the often required social interaction.

Ana Varbanescu [1], who is persuing a PhD at Delft University, was kind enough to answer a number of questions I had with regard to the used hardware. This not only helped solving a few specific problems, but also gave me feedback on the way I was working on the project in general.

Finally I want to thank my parents. Without their continued support, both mentally as financially, completing my studies would have never been possible.

Contents

1 Introduction **15**

2 Tomography **17**
2.1 Basic X-ray physics 17
2.2 Basic tomography mathematics 18
2.3 Calculating W 20
2.4 Space considerations when storing W 21
2.5 Restoring the original from projections 23

3 Simultaneous Algebraic Reconstruction Technique (SART) **25**
3.1 Inner workings of SART 25
3.2 Performance of SART 26

4 The Cell Processor **29**
4.1 SART implementation on the Cell 30
4.2 Performance of manual SART implementation on the Cell 31

5 Automating parallel code generation LIACS-style: COMPAAN **37**
5.1 Simple COMPAAN example: matrix multiplication 37
5.2 Kahn Process Networks (KPNs) 40
5.3 Implementation of FIFOs 43
5.4 A practical consideration: FIFO sizes 46
5.5 Our first and most simple usage of the SPEs 49
5.6 Performance of this solution 50
5.7 Implementing a more advanced way of inter-process communication 50
5.8 Flushing operations 56

6 Conclusion **61**

A Description of files generated by CellCC **63**
A.1 types.h 63
A.2 ppu_buffers.h 63
A.3 PPU Makefile 64
A.4 spu_buffers.h 65
A.5 SPE Makefile 65
A.6 main.cc 66
A.7 SPEx.cpp 66

B Usage of code base **67**
B.1 sart/ 67
B.2 sart/writematrix/ 68
B.3 sart/bmp2img/ 68
B.4 sart/componized/sanlp_pthreads/ 68
B.5 sart/cell/ 68

List of Figures

1.1 X-ray image of male skull. All depth information is lost. 15

1.2 More helpful dental X-ray image where the detector is placed inside the mouth. Now teeth can be examined separately. 16

2.1 X-rays are produced at the source point and move through the object to be measured by a detector placed on larger detector plate. A part of the ray through the object is labeled 'ds', this represents the infinitesimal small part of the ray where the intensity is reduced because of absorption. 18

2.2 Schema of the type of projections used: image (center) is 2D, the projections consist of a finite number of rays (arrows from lower left) and detectors (upper right) are ordered one-dimensionally. The graph above the detectors depicts the measured line integrals. 18

2.3 Typical projection: an object (here shown as a gray blob) is placed on a raster of n by n squares numbered from top left to lower right (x,y-information is no longer used). Every square (labeled x) will contain a value between 0.0 (nothing there) and 1.0 (highest density). Also shown are a number of strips drawn out by projecting rays. All strips have the same width and every entry of $\vec{p}$ will contain the total overlapping area of the object and the corresponding strip. One polygon has been highlighted to explain the calculation of the weight matrix (see text). . . 19

2.4 A number of ways in which a square and a strip can overlap: (a) en (f) are special cases, with an overlapping area of zero and one respectively. The rest show intermediate possibilities (all other cases can be constructed by rotating and/or mirroring these examples). Number of utilized vertices (see text): (a): 0; (b): 3; (c): 4; (d): 5; (e): 6; (f): 4. 20

2.5 Steps of algorithm used to calculate overlapping area of strip and square. Explanation (see text): (a): finding of square corners inside of strip and intersections of rays and square edges; (b): when utilized points are found (are sorted clockwise from top), create corresponding triangles (numbered in creation order); (c): calculate area of triangles via basic vector arithmetic. 21
2.6 Pseudo code implementation of algorithm to calculate overlap of square and strip. CalcArea(A, B, C) is shown in Image 2.5(c). 22
2.7 Storing sparse matrix more space-effectively. (a): all entries not given of column 11 are zero, rows are numbered. (b): matrix now contains only 3 rows (maximum number of non-zero entries in columns of original matrix), yet number of columns is doubled. Column 22 and 23 ($2n$ and $2n + 1$, with $n = 11$) contain original row number and stored data, respectively. 24
2.8 Algebraic operation performed when $\vec{x}$ is projected in d directions. Both $\mathbf{W}$ and $\vec{p}$ consist of d vertically ordered items. 24

3.1 SART algorithm (see text). 26
3.2 SART in action: starting with a completely black image $\vec{x}$, the original image is restored. First row: original image (which was projected in 40 directions and $\vec{x}$ at beginning of restoration, which is equal to $\vec{0}$. Second row: first cycle, after which some outlines start to become visible. Third row: $\vec{x}$ gets closer to original image. Fourth row: $\vec{x}$ at end of third, fourth and fifth cycle. 27
3.3 Average error of $\vec{x}$ with respect to number of iterations used. The graph describes same run as figure 3.2, where every cycle consists of 40 iterations. The average error moves slowly to 0.07, which is acceptable in most circumstances. 28

4.1 Overview of Cell CPU. On the left the PPU is shown and its two threads. On the right are its 8 SPUs, all connected by a bus to the other SPUs and the PPU. Note that SPUs don't have direct access to main memory, but do have a local storage. Also shown are the four inboxes and one outbox of every SPU. These are used to communicate via mail messages. 30
4.2 PPU pseudo code of implementation of SART on the Cell. See text for an in depth explanation. 'SartCellSPU' is a pointer to the code of figure 4.3. 'Concatenate(a, b)' returns a vector consisting of all entries of a followed by all entries of b. 32
4.3 Code executed on the SPU. We assume the variable 'this' stores the information for the SPU on which this code runs. Except for the mailbox communications with the PPU, the looped code is the same as in figure 3.1. 33
4.4 Relation between achieved speed-up of our parallel code compared to the serial code and image size. A polynomial trend line which fits the data is shown. 35

4.5 Speed-up of parallel code compared to code running on regular AMD64 3800+ (2.01 GHz) processor. Maxes out at about 7. The polynomial trend line, which can be regarded as an average, reaches 6.5. 36

5.1 MATLAB code that loads two 32 by 32 matrices, multiplies them and stores the result. 39
5.2 Implementation of MultiplyAndAdd and LoadZero. Note that this code isn't used by COMPAAN, but will be written in the target language. 40
5.3 Data dependencies when calculating the upper left element of matrix $\mathbf{C}$. First row of $\mathbf{A}$ is needed in its entirely, first column of $\mathbf{B}$ is also needed. Because matrices $\mathbf{A}$ and $\mathbf{B}$ are read in row-major order, a large part of $\mathbf{B}$ needs to be read before the first column is read (here shown in light gray). 40
5.4 Two simple Kahn Process Networks showing structure of this type of networks. . 41
5.5 KPN produced by COMPAAN for our matrix multiplication code. Node 3 isn't connected to any other nodes (as expected) and the outputs of node 1 and 2 need to be reordered inside node 5, this action is shown here in gray (see text). Also note that FIFO 1 connects node 5 to itself, because the vector product of a row of A and a column of B is calculated by adding multiple values to the same entry of C, which is therefore used as an input of node 5 and as an output. 42
5.6 Code executed by node 1. 'LoadA' needs to be implemented by the programmer in the target language (in this case $C++$) and stores the return value in the last argument, 'write' is a method defined by the environment that finds a FIFO (first argument) and writes a value to it (second argument), blocking if the FIFO is full. 43
5.7 Code running on node 5. It is a lot more complex than node 1, because it uses three input FIFOs, of which two are buffered. The code that reads the input variables is given in figure 5.8. 44
5.8 Code responsible for reading input variables of node 5. 45
5.9 Initialization code of FIFO. Declares and initializes all local variables of a FIFO. 46
5.10 'WriteToken'. The mutex is locked, a check is performed to see if there is room in the buffer, the token is written, some bookkeeping is performed, the mutex is released, and when a reader is waiting, it is signaled that the buffer is no longer empty. 47
5.11 Code called to read a token. Almost the same as the 'WriteToken' code, only token is read from buffer, instead of written to it. See text for complete discussion of both methods. 48
5.12 A simple KPN with two nodes and two FIFOs. 49
5.13 KPN of figure 5.12 put on both the PPU and on the SPEs. A' and D' act as service threads that delegate all calculations to A and D, respectively, which are located on the SPEs. A and D use their serving threads to access the FIFOs. . . . 49

5.14 Service thread on PPU side. This thread runs while the thread on the SPE side runs. It monitors the outbox of its designated thread. See the text for a complete discussion. 51
5.15 Implementation of the three methods used by the code running on an SPE. All actions are implemented by sending and receiving the required mail messages. . . 52
5.16 A producer-consumer KPN. This network is used to measure the performance of the two protocols developed during this project. By moving the sending and receiving nodes to different Cell units the speed of all communications can be determined. 52
5.17 '/proc/cpuinfo' for the PlayStation 3. The two hardware threads are shown as separate processors (values given: index, description, clock speed and revision). Important to us: the 'timebase' value, it gives the number of increments of the 'time base' register per second. 53
5.18 Contents of 'cpuinfo' for the Cell simulator. It has a smaller timebase value (to lower the strain on the simulator), but does run on the theoretical speed of 3.2 GHz. 53
5.19 Our first, service-thread-based, solution's performance. All connections were tested for 2^{10} (1024), 2^{20} ($1024 \cdot 1024$), and 2^{27} ($128 \cdot 1024 \cdot 1024$) tokens and the required execution time was used to calculate the effective number of tokens per second. 54
5.20 Usage of DMA to offer a FIFO connection between two elements of the Cell processor. Steps 1 through 4 copy values to the buffer in element 1. Step 5 consists of sending a message to the reader to tell it the buffer is full, so it can start the DMA operation (step 6). After this is finished, a message is sent back to tell the buffer has been copied and can again be written to. 55
5.21 A 4-node KPN running on the PPU and two SPEs. All connection types are featured; 1: PPU to PPU, 2: PPU to SPE, 3: SPE to SPE, 4: SPE to PPU, 5: PPU self loop, and 6: SPE self loop. We needed to deal with all of them separately. . . 56
5.22 Performance for our second implementation, which is DMA based. The buffer size is 1024 tokens. 57
5.23 Comparison of the two discussed methods. Method 1 is service-thread-based, method 2 is DMA based. The third column gives the speed-up of method 2 over method 1. Only the PPU to PPU method has comparable performance, because this functionality is implemented the same. 57
5.24 A simple KPN terminating in a deadlock situation if no flushing is performed. After node 1 has written a token, but doesn't flush FIFO 2, both nodes are blocked in read state. 58
5.25 Effect of continuously flushing on performance of communicating 2^{27} tokens. Comparison is made to method 1 (service-threads) and method 2 (fully buffered DMA). The used buffer size is 1024 tokens. 59

5.26 Performance of method 2 when continuously flushing the DMA buffers compared to method 1 and the original method 2. The buffer size is reduced to 4 tokens, to see what it does to the required time to communicate compared to the 1024 buffer size. 59

Chapter 1

Introduction

Non-intrusive imagining is a very important technique with a large number of applications. A well known example is X-ray imaging. Using this technology it is possible to diagnose a large number of illnesses. Surgeons and dentists wouldn't be able to do their job as well without X-ray technology.

Standard X-ray applications have one big problem. Just like when a picture is taken with normal light, it loses depth information. This means the doctor has to place the patient between the X-ray source and detector in such a way that the needed information is acquired. Figure 1.1 gives an example of this problem: if a dentist was interested in the state of the patients teeth, this picture wouldn't be very helpful, because the teeth are projected on the same place. Figure 1.2 is taken with the detector inside the mouth, allowing each tooth to be inspected separately.

Figure 1.1: X-ray image of male skull. All depth information is lost.

Figure 1.2: More helpful dental X-ray image where the detector is placed inside the mouth. Now teeth can be examined separately.

One solution to this problem will be dealt with in this report. So-called CT-imaging allows a medical operator to restore the original 3D information. CT stands for Computed Tomography and makes use of several images of the same object, but from different angles. As we will see, there are algorithms to do this. We will implement one on different architectures to be able to say something about its performance and used hardware.

Before we take a more in-depth look at the used algorithm and other details, we will have to deal with the physics encountered when taking an X-ray image.

Chapter 2

Tomography

2.1 Basic X-ray physics

X-ray imaging works because different materials absorb the rays in different amounts. Absorption coefficients give the percentage of radiation that is absorbed. Following figure 2.1, the ray starts at the source and when it enters the object it begins to lose intensity. The detected intensity at the receiving end is then used to calculate the total material encountered.A ray consists of photons which, because of their very short wavelength, act as particles.

When we measure a reduced intensity of an X-ray, this means that a number of photons were absorbed by the material. So a detector is nothing more than a photon counter. If we now look at a small part of the ray when it is inside the object (labeled 'ds' in figure 2.1) we can say a number of things about it. If ds is homogenous (the ray goes through the same material), the intensity at the end of ds will be:

$$I = I_0 \cdot e^{-\mu \cdot |ds|} \tag{2.1}$$

With I the intensity at the end, I_0 at the beginning, μ the attenuation factor and $|ds|$ the length of ds. Of course μ isn't the same for every ds, we can only assume this if ds is very small. This happens if we use a line integral over the beam:

$$I = I_0 \cdot e^{-\int_0^s \mu(x)dx} \tag{2.2}$$

$$-ln(\frac{I}{I_0}) = \int_0^s \mu(x)dx \tag{2.3}$$

We know I_0 and measure I, this allows us to calculate the line integral and use it later on when we restore the image. This will be dealt with next.

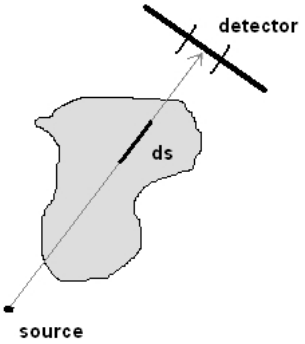

Figure 2.1: X-rays are produced at the source point and move through the object to be measured by a detector placed on larger detector plate. A part of the ray through the object is labeled ‘ds’, this represents the infinitesimal small part of the ray where the intensity is reduced because of absorption.

2.2 Basic tomography mathematics

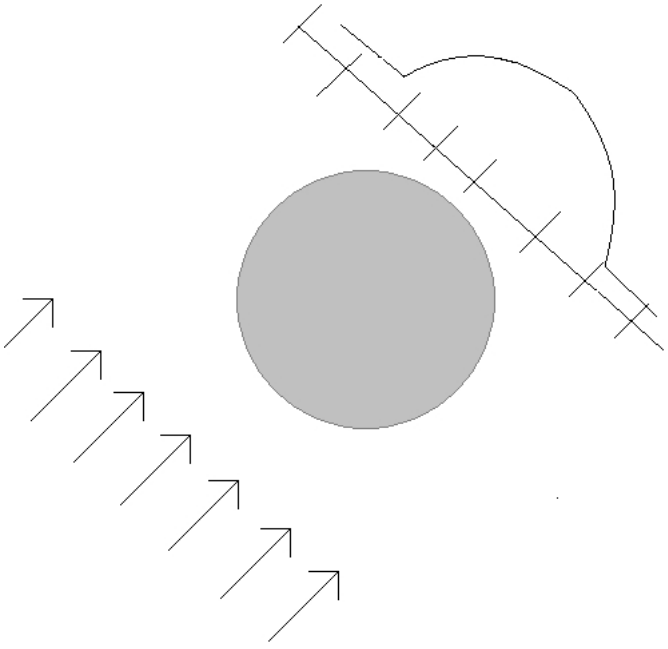

Figure 2.2: Schema of the type of projections used: image (center) is 2D, the projections consist of a finite number of rays (arrows from lower left) and detectors (upper right) are ordered one-dimensionally. The graph above the detectors depicts the measured line integrals.

We will now take a look at the math behind a projection (see figure 2.3). To reconstruct a slice of the object, we have placed a grid on it. Since all considered objects are either circle shaped or close to it (for example a human head), its height and width differ very little and a square grid is acceptable. We assume a finite number of rays whose tracks have certain width (each one draws out a strip). A value measured on the receiving end (a detector) is equal to the area of the object on the respective strip. Figure 2.3 shows just one direction of rays (and only partial, when running the algorithm the entire grid is covered by strips), we shall later on see that there are in fact a large number of projection angles, each increasing the number of measurements and quality of the reconstructed image.

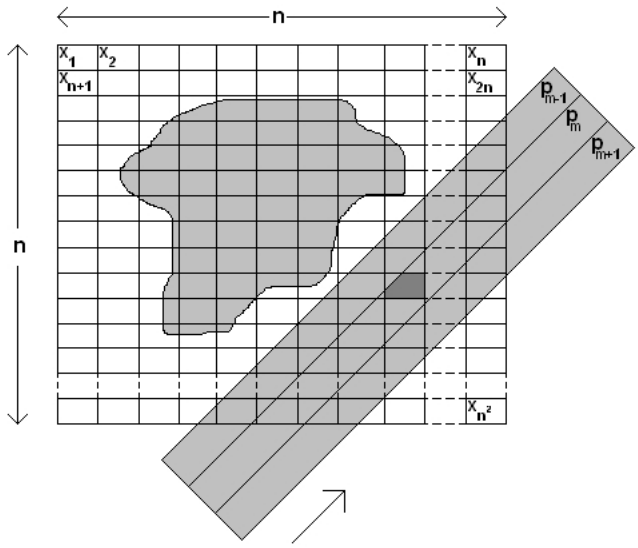

Figure 2.3: Typical projection: an object (here shown as a gray blob) is placed on a raster of n by n squares numbered from top left to lower right (x,y-information is no longer used). Every square (labeled x) will contain a value between 0.0 (nothing there) and 1.0 (highest density). Also shown are a number of strips drawn out by projecting rays. All strips have the same width and every entry of $\vec{p}$ will contain the total overlapping area of the object and the corresponding strip. One polygon has been highlighted to explain the calculation of the weight matrix (see text).

This type of projection can be described by a matrix operation. The original image is stored as a one dimensional vector $\vec{x}$. The projection matrix is called $\mathbf{W}$, and the projection will be stored in $\vec{p}$. $\vec{x}$ contains n^2 elements, $\vec{p}$ contains j elements (one for each detector). The following operation describes the projection:

$$\mathbf{W} \cdot \vec{x} = \vec{p} \tag{2.4}$$

To make this a valid matrix operation, $\mathbf{W}$ needs to have n^2 columns and j rows. An entry

of $\mathbf{W}$, let a be its row index and b its column index, is defined as follows. The entry of $\vec{x}$ with index b is multiplied with $\mathbf{W}_{a,b}$ and added to entry with index a of p. This shows that the $\mathbf{W}_{a,b}$ will resemble the 'weight' of square b when calculating the value of the projection of strip a. When a square is not covered by a strip, the respective entry will be zero. If it is covered, the value represents the overlapped area. It will have the maximal value of 1.0 when the square is completely inside the strip. When it is only partially covered some basic arithmetic is needed to calculate its value.

2.3 Calculating W

A strip and a square can overlap in number of ways. One way is highlighted in figure 2.3 and figure 2.4 shows a few more ways. We developed an algorithm which calculates the overlapping area of a square and a strip, given some basic properties of both. We will show the method we used by applying it on a general case and later show the pseudo code which can deal with all cases (following figure 2.5).

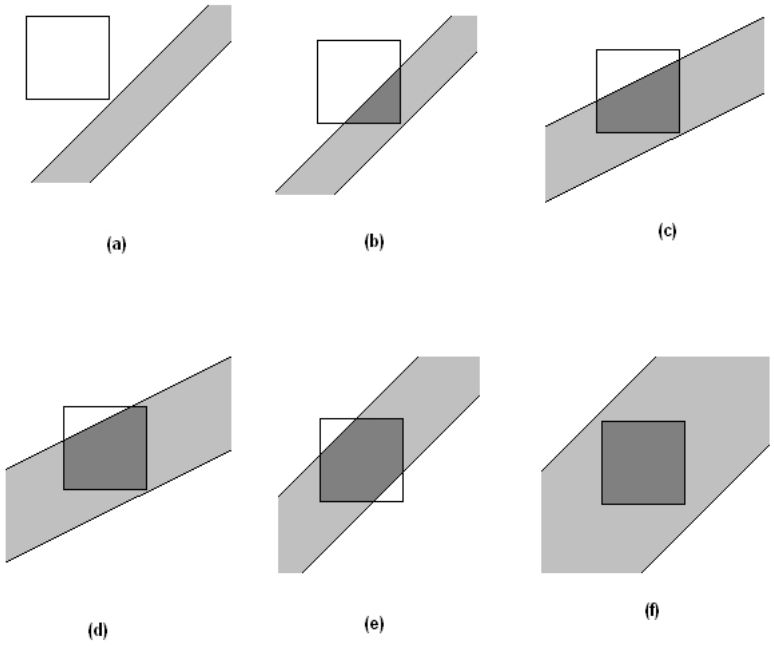

Figure 2.4: A number of ways in which a square and a strip can overlap: (a) en (f) are special cases, with an overlapping area of zero and one respectively. The rest show intermediate possibilities (all other cases can be constructed by rotating and/or mirroring these examples). Number of utilized vertices (see text): (a): 0; (b): 3; (c): 4; (d): 5; (e): 6; (f): 4.

We start by defining x_0, a vertical line. It will be to the left of the square. We calculate the

intersection points of this line with the rays. Then we trace all four square corners parallel to the rays (we use their tangent for that) and keep the ones that are projected on x_0 between the points where the rays intersected with it. After that we add the intersections between the square edges and the rays. In our example that produces six points (two square corners and four ray-edge intersections), we call these 'utilized points'. These form corners of a polygon of which we need to calculate the area. Sub image (b) shows what happens next: all found points are sorted clockwise and used to build triangles. One point (one closest to the top (and to the right)) will be used in all triangles. The areas are calculated (see figure 2.5(c)) and summated.

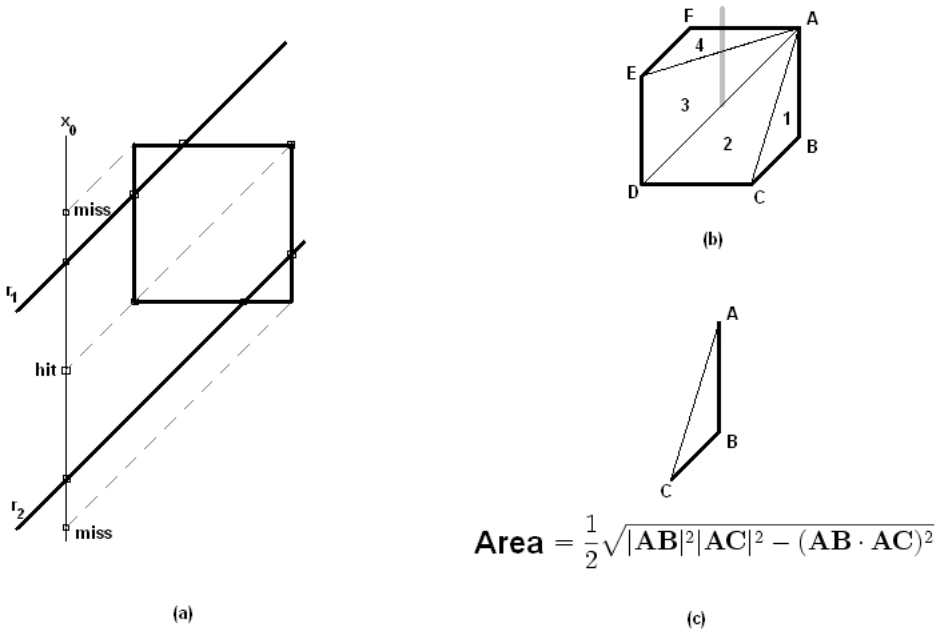

Figure 2.5: Steps of algorithm used to calculate overlapping area of strip and square. Explanation (see text): (a): finding of square corners inside of strip and intersections of rays and square edges; (b): when utilized points are found (are sorted clock-wise from top), create corresponding triangles (numbered in creation order); (c): calculate area of triangles via basic vector arithmetic.

The returned value (see figure 2.6) will be stored in the ($\mathbf{W}_{stripnumber,pixelnumber}$).

2.4 Space considerations when storing W

All entries of a column of $\mathbf{W}$ enumerate the shared areas with all strips for just one pixel. Since only a very small number of strips touches one particular square, most entries of this column will be zero. And because a projection matrix grows to extreme sizes very rapidly (a 256 by 256

```
tang := tangent of rays
Corners := Set containing corners of square
Edges := Set containing edges of square
UtilPoints := empty Set

P := intersection of x0 and r1
Q := intersection of x0 and r2

foreach point s in Corners do
  y0 := s.y  (s.x  x0) * tang
  if((y0 >= P.y) and (y0 < Q.y)) then
    UtilPoints.add(s)
  end if
end foreach

foreach line e of Edges do
  inter := intersection of e and r1
  if(inter ison e) then
    UtilPoints.add(s)
  end if
  inter := intersection of e and r2
  if(inter ison e) then
    UtilPoints.add(s)
  end if
end foreach

SortClockwise(UtilPoints)

A := UtilPoints.first
UtilPoints.RemoveFirst()

total_area := 0
for point_index := 1 to (UtilPoints.size - 1) do
  total_area := total_area + CalcArea(A, UtilPoints[point_index],
                                      UtilPoints[point_index + 1])
end for

return total_area
```

Figure 2.6: Pseudo code implementation of algorithm to calculate overlap of square and strip. CalcArea(A, B, C) is shown in Image 2.5(c).

pixels image covered by 363 strips needs to be described by a matrix containing over six billion entries) we decided to store the matrix in a way that would exploit its sparseness.

Instead of storing complete columns, we effectively double the number of columns where every row pair consists of the original row number and the stored value (see figure 2.7). Storing a matrix for a 256 by 256 image covered by 363 strips now takes only 262 thousand entries.

2.5 Restoring the original from projections

As explained in the introductory chapter, a projection is performed from multiple angles. Until now we only reviewed one projection (from one direction), but we will now see that this process can be easily repeated to cover projections from more than one angle. All what is needed is extending both $\mathbf{W}$ and $\vec{p}$. The number of rows of both $\vec{x}$ and $\mathbf{W}$ are multiplied by the number of projection directions. The new $\mathbf{W}$ will consist of several original projection matrices stored above each other (see figure 2.8).

When we have calculated $\mathbf{W}$ and measured p, we could solve equation 2.4 to obtain $\vec{p}$, but due to the enormous size of $\mathbf{W}$ this is not a realistic option. Also note that this matrix is in general not square (n^2 is in practice always larger than the number of projections multiplied by the number of projection angles) so performing a matrix inversion of $\mathbf{W}$ isn't possible (the solution would be a multi-dimensional solution space). A last reason why an algebraic solution won't work is measurement errors. As with all practical measurements, one can never ignore the potential for errors. This will make solving $\vec{x}$ impossible. All these problems demand an other approach.

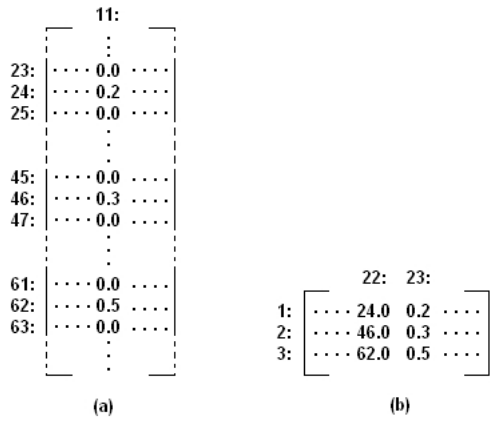

Figure 2.7: Storing sparse matrix more space-effectively. (a): all entries not given of column 11 are zero, rows are numbered. (b): matrix now contains only 3 rows (maximum number of non-zero entries in columns of original matrix), yet number of columns is doubled. Column 22 and 23 ($2n$ and $2n+1$, with $n = 11$) contain original row number and stored data, respectively.

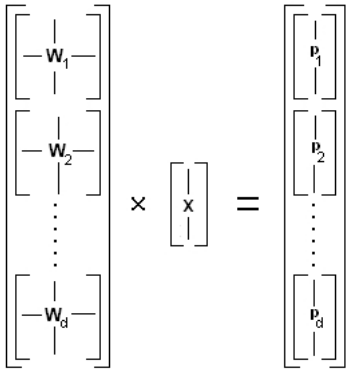

Figure 2.8: Algebraic operation performed when $\vec{x}$ is projected in d directions. Both $\mathbf{W}$ and $\vec{p}$ consist of d vertically ordered items.

Chapter 3

Simultaneous Algebraic Reconstruction Technique (SART)

A number of different algorithms have been proposed to solve large scale tomography problems. We use 'SART' and before we will show its performance, it will be explained. 'SART' is an algorithm that has the best result when used with images when no foreknowledge of it is available. If there is some knowledge available about the image to be reconstructed, other algorithms are helpful, such as discrete tomographic algorithms (used when the object consists of only a few different material types) and TV-minimalisation (if it is known the image has large areas of one type of material).

3.1 Inner workings of SART

SART [2] is an iterative algorithm: it performs the computation of $\vec{x}$ in a number of distinct steps and will, on average, get closer to the correct answer with every step. Instead of one large matrix $\mathbf{W}$ and $\vec{p}$, the data is stored as before: separate for each projection direction. Every direction amounts to its own iteration, where a number of basic operations are performed to calculate $\vec{x}$.

A projection of $\vec{x}$ is simulated and stored in $\vec{u}$. This vector's dimensions will be the same as $\vec{p}$, but its values will be different. The difference is be stored in $\vec{err}$ ($= \vec{p} - \vec{u}$). This gives the error of each strip, not of separate pixels. Here SART assumes all pixels caused the projected error according to the area covered by the strip currently considered (this is stored in $\mathbf{W}_{stripindex,pixelindex}$). The entry of $\vec{delta}$ for this pixel will be increased in the following way:

$$\vec{delta}_{pixelindex} = \vec{delta}_{pixelindex} + \vec{err}_{stripindex} \cdot \frac{\mathbf{W}_{stripindex,pixelindex}}{\vec{beta}_{stripindex}}$$

The added value needs to be divided by the total area of a strip (here stored in $\vec{beta}$), because a pixel on a larger strip contributes less to the projection error than one on a smaller strip.

After $\vec{delta}$ has been calculated and before it is added to $\vec{x}_{pixelindex}$, it needs to be divided

```
foreach iteration do
  foreach direction dir in Directions do
    W := GetWMatrix(dir)
    p := GetPVector(dir)
    beta := GetBeta(dir)
    gamma := GetGamma(dir)
    u := W * x
    err := p  u

    delta := 0
    for i := 1 to (n*n) do
      for j := 1 to strips do
        delta[i] := delta[i] + err[j] * W[j][i] / beta[j]
      end for

      x[i] := x[i] + delta[i] / gamma[i]
    end for
  end foreach
end foreach
```

Figure 3.1: SART algorithm (see text).

by $\vec{gamma}_{pixelindex}$. $\vec{gamma}_{pixelindex}$ stores the total coverage of each pixel for this direction. Some implementation leave this factor out, beceause it almost always equals 1.0, but we left it in, to be able to deal with projections that don't cover all pixels.

See figure 3.1 for SART in pseudo code.

3.2 Performance of SART

Before considering our specific implementations of SART, we will first give some typical results of this algorithm. Figure 3.2 shows a number of screenshots of $\vec{x}$ during a run of the algorithm. The image is 240 by 240 pixels, covered by 40 projection directions, each consisting of 340 strips. The restoration process is seen to move clockwise through the image, where at the end of every cycle $\vec{x}$ gets closer to the original image.

It took an AMD with a clock speed of 2 GHz 2 hours to perform the first 600 iterations, after which the minimum error is reached (see figure 3.2). This is why it is interesting to implement it on a faster machine so the computation time can be reduced.

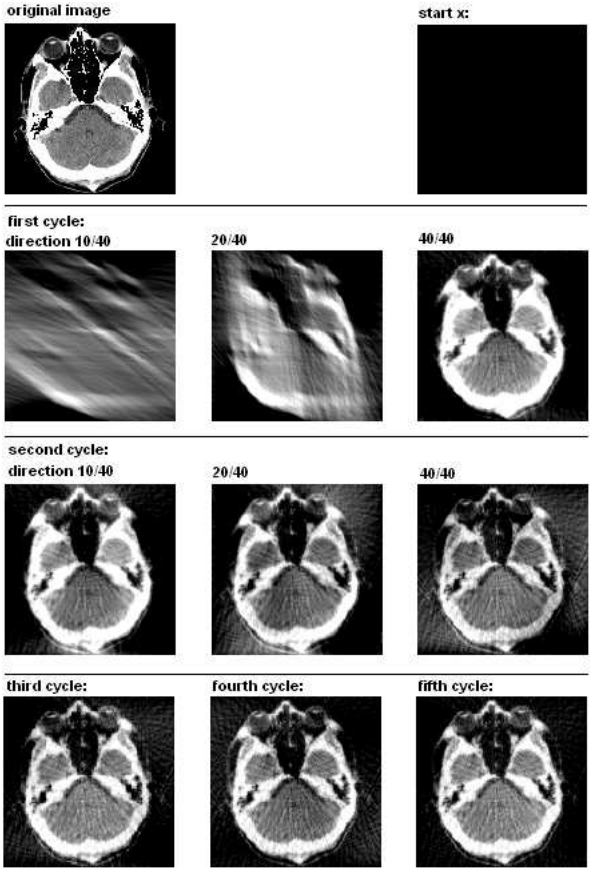

Figure 3.2: SART in action: starting with a completely black image $\vec{x}$, the original image is restored. First row: original image (which was projected in 40 directions and $\vec{x}$ at beginning of restoration, which is equal to $\vec{0}$. Second row: first cycle, after which some outlines start to become visible. Third row: $\vec{x}$ gets closer to original image. Fourth row: $\vec{x}$ at end of third, fourth and fifth cycle.

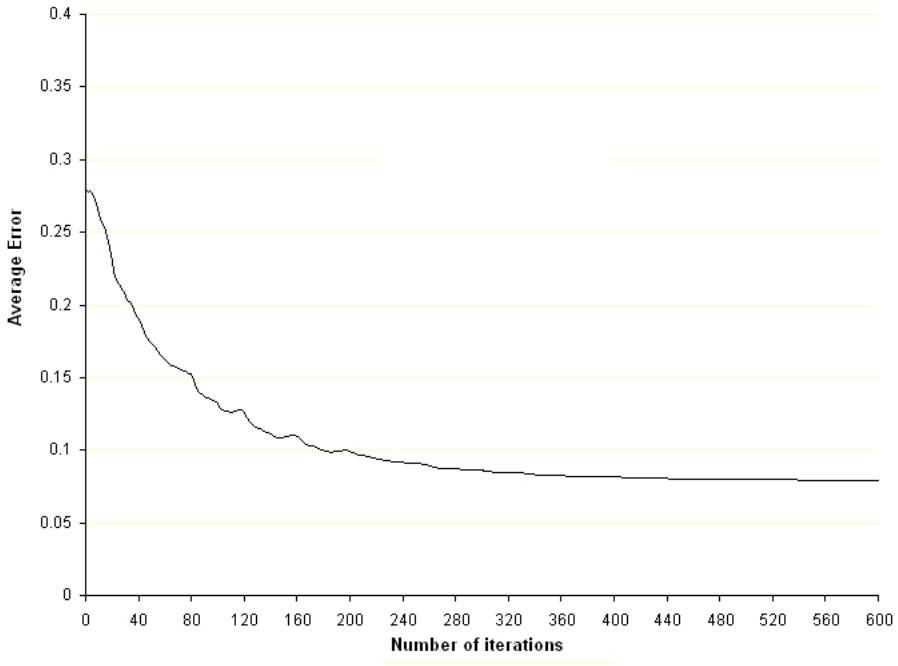

Figure 3.3: Average error of $\vec{x}$ with respect to number of iterations used. The graph describes same run as figure 3.2, where every cycle consists of 40 iterations. The average error moves slowly to 0.07, which is acceptable in most circumstances.

Chapter 4

The Cell Processor

IBM teamed up with Sony and Toshiba to form STI, a joint venture to develop a new type of processor, the Cell [3][4]. This processor consists of a number of distinct units (see figure 4.1):

- A regular PowerPC unit with two threads implemented in hardware.
- Eight SPEs (Synergetic Processing Elements), also called SPUs, with 256 Kb of local storage that work like mini-CPUs. They can execute specially compiled code loaded on them from the PPU. They don't have direct access to the main memory, this is done via the PPU.

This setup offers programmers a chance to write parallel programs without paying the high price of most parallel systems. The theoretical computing power of the Cell is a terraFLOPS. A typical desktop CPU sold at that time has about ten gigaFLOPS.

Parallel processors offer extreme computing power, but require a new way of programming. Before we look at the way Leiden University tries to automate parallel code generation, we will show how we manually implemented SART on the Cell.

Because the Cell is made up of different units, communication is important. We used the following channels:

- Direct Memory Access (DMA): here an SPE accesses the main memory directly. Data is copied from and to the main memory from the SPE's local storage. There are some requirements: the maximum size copied is 16 Kb and needs to be a multiple of 128 bytes and the address needs to be quad-word aligned.
- Mailboxes: the SPEs can write a 32-bit message to their outbox. The PPU and other SPEs can read it. Every SPE also has one inbox with four slots, which can be written to by all other units. If a subunit tries to write to a full message box or read from an empty one, execution halts until a slot is free or a message is available, respectively.

We used the following operations:

- *StartSPU(spu, program)*: starts SPU *spu* with program *program*.
- *WaitUntilFinished(spu)*: blocks until an SPU is finished.
- *WriteToMailbox(spu, data)*: writes a message with value *data* to the inbox of SPU *spu*. If the mailboxes are full, this call blocks until a slot is free.
- *ReadFromMailbox(spu, data)*: reads from the outbox of an SPU, blocks if no messages in outbox. A read removes the message from the outbox.
- *WriteToOutbox(data)* (SPU only): stores *data* in local outbox. This call will block if outbox is full.
- *ReadFromInbox(data)* (SPU only): reads *data* from local inbox, blocks if empty. A read removes the message from the inbox.
- *DMA_read(PPU-name, SPU-name)* (SPU only): copies data from PPU to SPU.
- *DMA_write(SPU-name, PPU-name)* (SPU only): copies data from SPU to PPU.

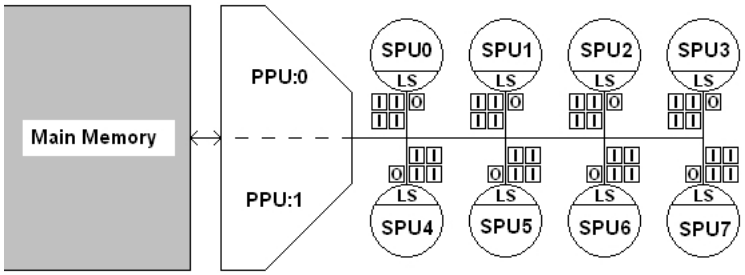

Figure 4.1: Overview of Cell CPU. On the left the PPU is shown and its two threads. On the right are its 8 SPUs, all connected by a bus to the other SPUs and the PPU. Note that SPUs don't have direct access to main memory, but do have a local storage. Also shown are the four inboxes and one outbox of every SPU. These are used to communicate via mail messages.

4.1 SART implementation on the Cell

The main aim of this project is to find a way to implement SART on an other type of hardware, so the restoration process will be quicker. We choose to implemented it on the Cell to find out if

this was a realistic option. We needed to add some parallelization to the code, to allow it to run on multiple SPUs.

The general idea is to split $\vec{x}$ over the SPUs and use the PPU for managing the threads and synchronizing the data. This means that every SPU only has to store an eighth of $\mathbf{W}$ in local storage. Also $\vec{beta}$ and $\vec{gamma}$ can be stored separately over the SPUs. Because the actions of PPU and SPUs are strongly intertwined we will deal with them at once, so please follow listing 3 and 4 carefully.

The PPU begins with preparing the data used by the SPUs and starting them. The PPU will then try to read a message from all their outboxes, which will be blocked until they write . They will now read in their parts of $\mathbf{W}$, $\vec{beta}$, $\vec{gamma}$, and the complete $\vec{p}$ into local storage using DMA. Then they will calculate the error caused by their pixels by projecting their $\vec{x}$ and subtract it from $\vec{p}$. The calculated $\vec{err}$s of all SPUs then need to be added, so will be DMA-ed to the PPU memory.

The PPU is told by the SPU the error vectors are calculated by writing a dummy value into the outbox. When all eight SPUs have done this, the PPU will add all error vectors and store it in $\vec{err}_{total}$ so the SPUs will be able to retrieve it.

Now the PPU will send a message to all SPUs and they will wake up, copy the total error vector and continue the SART algorithm, which will be the same as the original sequential code.

When the SPUs are done, they copy their values of $\vec{x}$ to main memory and the PPU copy the values in the final $\vec{x}$.

We chose this approach because it limited the number of DMAs and mail messages. By copying all required data to the SPEs at the beginning, we only need to send mail messages so the PPU knows an SPE has finished a phase of its calculations. All SPEs are loaded symmetrically, so none of them is waiting while the others are working. And except from the short periods during which the SPEs are waiting for the PPU to add the $\vec{err}$ vectors all SPEs are busy calculating.

4.2 Performance of manual SART implementation on the Cell

We will now look at how good this code is compared to the serial implementation. The cheapest way to obtain a working STI Cell configuration is by buying a Sony PlayStation 3 [5]. This latest generation gaming console costs about 600 euros and only has six SPUs available (two are used for OS security). Because this project is part of an initial exploration of the Cell's power and it wasn't known how likely it was we were to pursuit this architecture very long, we decided to use the offered simulator instead. This simulator can be downloaded for free from IBM's website [6] and allows a programmer to test and debug his software even when this hardware is not available. But as always is the case with simulators, executing code on it is much slower than on the real thing.

The simulator offers a number of settings on how close the inner workings of the CPU needs

```
foreach spu in SPUs do
  W[SPU] := CalcLocalW(spu)
  beta[SPU] := CalcLocalBeta(spu)
  gamma[SPU] := CalcLocalGamma(spu)

  StartSPU(spu, SartCellSPU)
end foreach

foreach iteration do
  foreach direction dir in Directions do
    foreach spu in SPUs do
      ReadFromMailbox(spu, dummy)
    end foreach

    err_total := 0
    foreach spu in SPUs do
      err_total := err_total + err[spu]
    end foreach

    foreach spu in SPUs do
      WriteToMailbox(spu, dummy)
    end foreach
  end foreach
end foreach

foreach spu in SPUs do
  WaitUntilFinished(spu)
end foreach

x := 0
foreach spu in SPUs do
  x := Concatenate(x, x_local[spu]);
end foreach
```

Figure 4.2: PPU pseudo code of implementation of SART on the Cell. See text for an in depth explanation. ‘SartCellSPU’ is a pointer to the code of figure 4.3. ‘Concatenate(a, b)’ returns a vector consisting of all entries of a followed by all entries of b.

```
DMA_read(W[this], W)
DMA_read(p, p)
DMA_read(beta[this], beta)
DMA_read(gamma[this], gamma)

foreach iteration do
  foreach dir in Direction do
    q := W * x
    err := q - p

    DMA_write(err, err[this])
    WriteToOutbox(dummy);
    ReadFromInbox(dummy);
    DMA_read(err_total, err);

    delta := 0
    for i := 1 to (n * n) do
      for j := 1 to strips do
        delta[i] := delta[i] + err[j] * W[j][i] / beta[j]
      end for

      x[i] := x[i] + delta[i] / gamma[i]
    end for
  end foreach
end foreach

DMA_write(x, x_local[this]);
```

Figure 4.3: Code executed on the SPU. We assume the variable ‘this’ stores the information for the SPU on which this code runs. Except for the mailbox communications with the PPU, the looped code is the same as in figure 3.1.

to be imitated. The two we used are called ‘fast mode’ and ‘pipe mode’. They can be considered as the extremes of a spectrum: the first runs the code as fast as possible, only guarantying the functional correctness whereas the latter completely simulates the internal workings of the processor, which is much slower. ‘Pipe mode’ is called that way because it also simulates the internal pipeline that will show delays caused by, for example, missed branch predictions and cache misses. We use the fastest mode to test the code and the slowest to measure performance.

To test the performance of our code we put the simulator in pipe mode and let it run for a number of days on ever increasing image sizes. We started at 3x3 pixels and ended at 104x104. This gave us an idea of the cost of the added overhead. We measured the execution time of both the serial code (PPU only) and our parallel implementation to calculate the speed-up. Image 4.4 shows the results.

We found a clear relation between the size of the image and the achieved speed-up. An image of 45 by 45 pixels and larger is done faster by the parallel implementation. The reason for a speed-up smaller than one for smaller images (which means it takes more time to execute it with SPUs than with just the PPU) is because of the added synchronization code and DMA actions. Note that the reported speed up might paint a bit too positive image, because the serial implementation only uses part of the Cell, whereas the parallel implementation uses it completely, so the hardware costs for the parallel code are also higher for the SPU implementation. Since a typical image in medical applications has a size of about 256x256 or 512x512 pixels, which we didn’t test because of the execution time of the simulator, it is safe to assume we could also achive a speed-up in a practical setting.

When its performance is compared to a serial implementation on a regular desktop PC, the results are extremely satisfying. A speed-up of almost 7 is achieved, with 6.5 as a nice average. This means that a 2003 600 euros system (PlayStation 3) out-performs a 2007 800 euros pc by about seven to one, a very good result. There is only one hidden factor we haven’t taken into account yet, which might skew things the other way: development time. This will be dealt with in the next section.

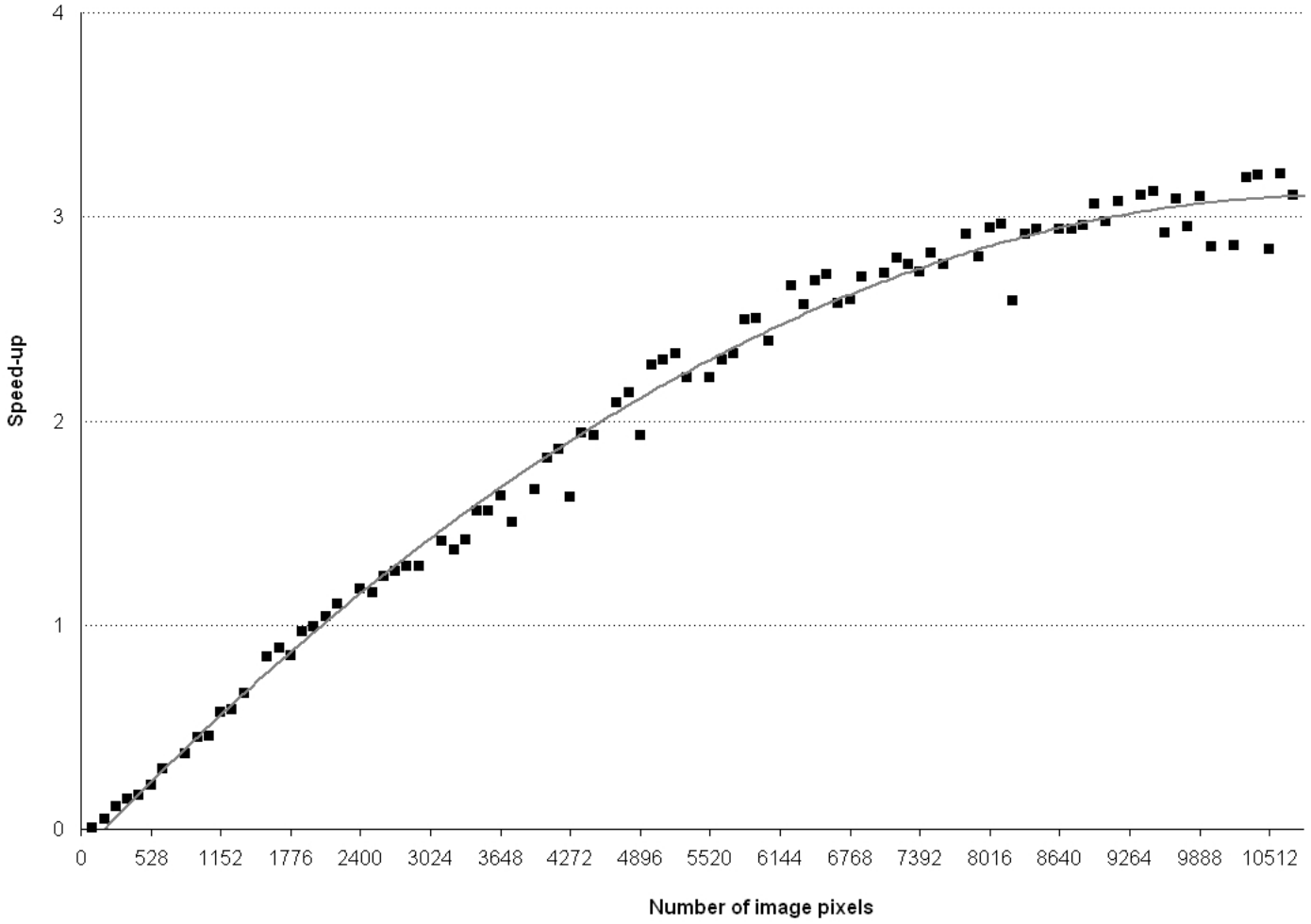

Figure 4.4: Relation between achieved speed-up of our parallel code compared to the serial code and image size. A polynomial trend line which fits the data is shown.

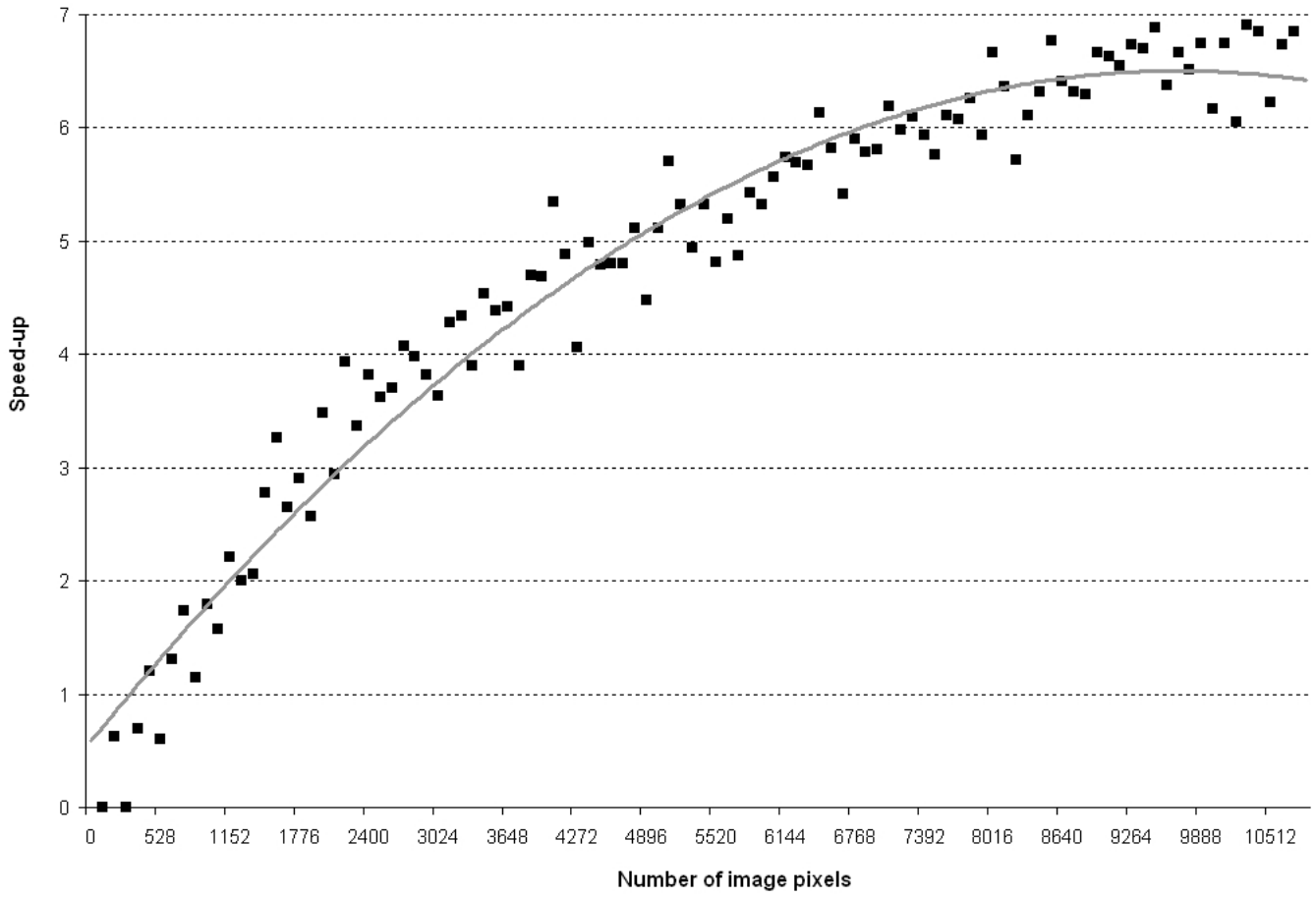

Figure 4.5: Speed-up of parallel code compared to code running on regular AMD64 3800+ (2.01 GHz) processor. Maxes out at about 7. The polynomial trend line, which can be regarded as an average, reaches 6.5.

Chapter 5

Automating parallel code generation LIACS-style: COMPAAN

Learning the Cell processor, coming up with the idea of how to balance the loads on the SPEs, writing the code, and debugging it took about two months. Once one is more experienced with the Cell, this can be brought down a bit, but writing parallel code still remains a notoriously hard job. It clearly started to dawn during the few last decades on system developers and other programmers that an automated process was needed. Many attempts have been made and I will focus on what has been done in this field by LIACS, Leiden University's computer science department.

We focus on structures found very often in the code requiring a speed-up: nested loops. Amdahl's Law tells us we need to focus on often executed code if we want to reduce the execution time of a program considerably. The body of a loop is executed very often and the deeper it is inside a collection of nested loops, the more we will gain by reducing its execution speed. In this case we will try to parallelize it as much as possible, to utilize the hardware we have.

Note that this will work the best when a lot of parallelization subunits are available. This document covers our exploration of the STI Cell, but we have been using this technique for the last couple of years already on FPGAs and are also interested in GPUs, which both offer more than one hundred units, compared to the eight offered by the Cell. Yet the SPEs have much more computing power than the elements of a typical FPGA implementation. This might be helpful later on, when we get to the implementation of SART using COMPAAN [7].

5.1 Simple COMPAAN example: matrix multiplication

We use a simple example to illustrate the strategy used by COMPAAN to parallelize code. Figure 5.1 shows a program that loads and multiplies two matrices and stores the result. Note that it is not given in a type of pseudo code, but in real matlab code. COMPAAN has matlab as its

input code, because this well-known programming language is very helpful when dealing with multidimensional arrays. Every executing statement has been labeled by a node number, this is useful when discussing it and also has some other uses, which will become more clear later on.

The first three nodes load the values of all entries of the matrices. This is typical for the initialization code needed by all COMPAAN programs, because all values of the used input arrays all need to be assigned. When the target code is generated, we need to edit this code so the correct values are loaded, which is going to be done by passing the row and column variables to the respective load functions.

In the main body the multiplication is performed. Every entry is set to zero and then the vector product of the appropriate row of $\mathbf{A}$ and column of $\mathbf{B}$ is calculated. Because COMPAAN is only interested in the order of the operations and the data dependencies (more about that later), it doesn't compile the calculations themselves. Those are hidden inside a function that is implemented later on in the target programming language, C++ in our case.

When the calculation is finished, the result is stored by calling *Pass*, that will see all values, which can then be used as a point where every value can be stored at a place where it can be written to disk, or used by other calculations.

What COMPAAN will look for, when compiling this file, is data dependencies. Nodes 1 through 3 normally will be executed $32 \cdot 32$ times before node 4 and 5 will be handled for the first time, although this is not strictly necessary. Node 4, for example, overwrites the value stored by node 3 without exception. This means node 3 doesn't need to be executed, or when it is executed, the resulting value doesn't need to be passed to the other parts of the program, because its value would be lost anyway.

Node 5 is more important. MultiplyAndAdd has three input values, which all need to be available when it is executed. In fact, when the loop containing this function (with iterator *index*) is about to be executed, the appropriate row of $\mathbf{A}$ and column of $\mathbf{B}$ need to be calculated. This means that not every instance of node 1 to 3 need to have been executed, just the ones that store at the locations of that row and column. This offers a helpful source of parallelization, because at some time almost every instance of node 4 and 5 can be executed before the initialization loop has finished.

In our case it could be done in the following way: PPU runs node 1, 2, and 3, because it is the only unit that allows file access, SPE 0 is running node 4, SPE 1 runs node 5, and the PPU again is running node 6. Now SPE 0 and 1 can start before the PPU is done, thus reducing the total execution time.

COMPAAN's job is to find the data dependencies and generate code that will take care of sorting it out. Figure 5.3 shows that the dependencies in this case are a bit too complex to just start node 4 and 5 as soon as some data is loaded. All entries of $\mathbf{A}$ and $\mathbf{B}$ are used multiple times and, from the initialization's point of view, out of order. This problem needs to be solved and will be the next point of attention.

```
for row=1:1:32,
  for column=1:1:32,
    %% Node 1
    [A(row, column)] = LoadA();
    %% Node 2
    [B(row, column)] = LoadB();
    %% Node 3
    [C(row, column)] = LoadC();
  end
end

for row=1:1:32,
  for column=1:1:32,
    %% Node 4
    [C(row, column)] = LoadZero();
    for index=1:1:32,
      %% Node 5
      [C(row, column)] = MultiplyAndAdd(C(row, column),
                                        A(row, index), B(index, column));
    end
  end
end

for row=1:1:32,
  for column=1:1:32,
    %% Node 6
    [SinkC(row, column)] = Pass(C(row, column));
  end
end
```

Figure 5.1: MATLAB code that loads two 32 by 32 matrices, multiplies them and stores the result.

```
function result = MultiplyAndAdd(c, a, b)
  result = c + a * b;
end

function result = LoadZero()
  result = 0;
end
```

Figure 5.2: Implementation of MultiplyAndAdd and LoadZero. Note that this code isn't used by COMPAAN, but will be written in the target language.

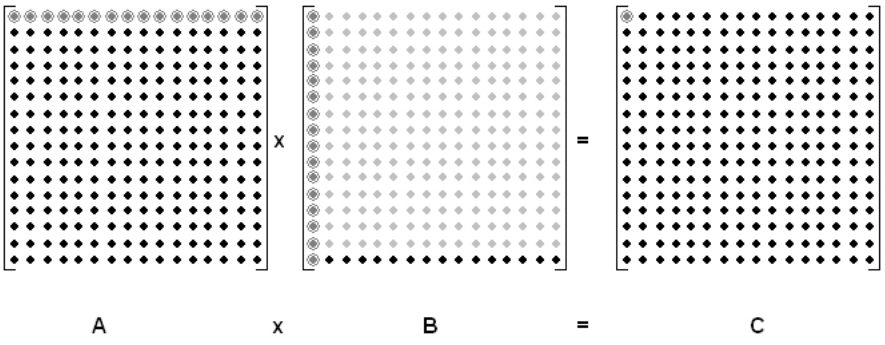

Figure 5.3: Data dependencies when calculating the upper left element of matrix $\mathbf{C}$. First row of $\mathbf{A}$ is needed in its entirely, first column of $\mathbf{B}$ is also needed. Because matrices $\mathbf{A}$ and $\mathbf{B}$ are read in row-major order, a large part of $\mathbf{B}$ needs to be read before the first column is read (here shown in light gray).

5.2 Kahn Process Networks (KPNs)

COMPAAN uses a subclass of Process Networks (PNs) called 'Kahn Process Networks' (KPNs), named after Dr. Gilles Kahn. Figure 5.4 shows two very simple KPNs to illustrate their general topography. All nodes are connected via FIFOs, queues implementing First-In-First-Out passing of items. Figure 5.4(a) shows a producer-consumer model, with node 1 being the producer. The KPN of figure 5.4(b) shows that nodes can have multiple input and output FIFOs, a property that will become helpful later on.

As we saw during our matrix multiplication example, COMPAAN needs to utilize the fact that statements, which we already named after their node number, can be started out of order compared to their serial counterpart. Node 4, for example could be started right away and node 5 when only a part of matrices $\mathbf{A}$ and $\mathbf{B}$ were loaded. COMPAAN produces a process for every node to be run from the start.

Every node without an input FIFO can start right away and when it has one or more output FIFOs, will eventually start writing tokens to them. Nodes that do have input FIFOs will start at the same time, but their reading operations will stall until a token is written to the FIFOs in question. If the order of writing to the FIFOs and reading from it are known on both sides, the nodes are able to execute in a parallel fashion. Figuring this out is COMPAAN's job.

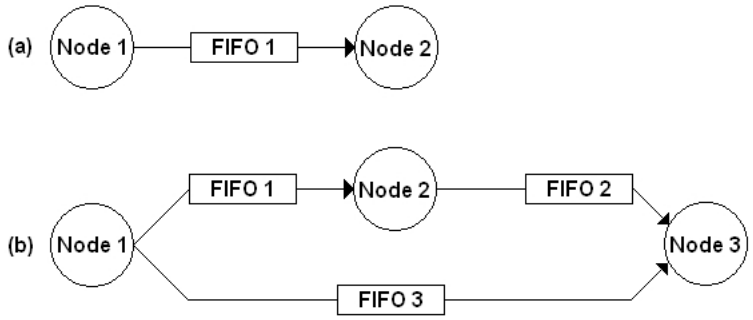

Figure 5.4: Two simple Kahn Process Networks showing structure of this type of networks.

Let's now take a look at our matrix multiplication example. We let COMPAAN compile the code of figure 5.1 and visualized the produced KPN, the result of which can be seen in figure 5.5. We see that node 3 has no input or output ports. This means it can be pruned from the KPN, something we already deduced ourselves when we previously discussed the matlab code. We also see that FIFO 1 connects node 5 with itself. This is because *C(row, column)* in node 5 is both used as an input and an output, so the produced value is needed during the following iteration.

A last remark deals with something we already saw in figure 5.3. Because the order in which the entries of matrices $\mathbf{A}$ and $\mathbf{B}$ are loaded are not the same as they are needed by node 5 and also because they will be used multiple times, they need to be reordered. COMPAAN therefore generates code that will read FIFO 3 and 4, but will offer the read tokens in the correct order.

Figure 5.6 shows the $C{+}{+}$ code running on node 1. It produces 32 times 32 tokens and places them in FIFO 3. The values of the tokens are defined by the *LoadA* method, to be implemented by the programmer. The value is returned in the last argument of the function, which is passed by reference. The programmer had to add the row and column index manually to the method invocation, because it wasn't in the original code. Because it is still in its development stage, code created by COMPAAN still requires some manual editing.

Figure 5.7 is a bit more interesting and shows what happens in node 5. As shown by figure 5.5, node 5 has four input FIFOs and one output FIFO. Input FIFOs 3 and 4 need a reordering of their output and FIFO 1's input and output are both connected to node 5. The code of node 5 has to take all that into account.

Before *MultiplyAndAdd* can be called, which is the name used for the function used in the original matlab code, the input variables need to be collected. The entry of $\mathbf{C}$ accessed by this instance is read from FIFO 2 when it is the first one and from FIFO 1 in all other cases. The values of the entry of $\mathbf{A}$ and $\mathbf{B}$ are read from the local buffer, that receives all tokens and stores them until the node 5 code reads them. Whenever node 5 is done with a part of the data (a row or a column) that is no longer needed, it tells this to the buffer, so it can purge the used data.

When the input data is read, the *MultiplyAndAdd* method can be called. As before, the last argument is passed by reference and will contain the output of the function. This function needs to be implemented by the programmer in $C++$. The output value, which is the current value of the entry in $\mathbf{C}$, then has to go into a FIFO. If the value of *index* is lower than 32, the value needs to be written to FIFO 1, so it will be read again by node 5. If *index* is equal to 32 (the vector product is calculated), it has to be written to FIFO 5, so it can be handled by node 6.

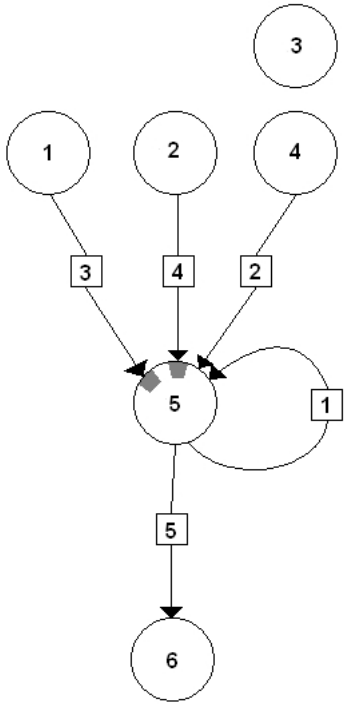

Figure 5.5: KPN produced by COMPAAN for our matrix multiplication code. Node 3 isn't connected to any other nodes (as expected) and the outputs of node 1 and 2 need to be reordered inside node 5, this action is shown here in gray (see text). Also note that FIFO 1 connects node 5 to itself, because the vector product of a row of A and a column of B is calculated by adding multiple values to the same entry of C, which is therefore used as an input of node 5 and as an output.

```
for(row = 1; row <= 32; row++)
{
  for(column = 1; column <= 32; column++)
  {
    LoadA(row, column, A_output_value);
    write(FIFO3, A_output_value);
  }
}
```

Figure 5.6: Code executed by node 1. 'LoadA' needs to be implemented by the programmer in the target language (in this case $C++$) and stores the return value in the last argument, 'write' is a method defined by the environment that finds a FIFO (first argument) and writes a value to it (second argument), blocking if the FIFO is full.

5.3 Implementation of FIFOs

COMPAAN generates the code for the nodes, but the rest of the environment (FIFOs, multi-threading and the like) has to be created by the programmer. A number of implementations already exist and COMPAAN can generate the code so it works with them. One is offered for $C++$ and is named 'YAPI' and also one for Java called 'Ptolemy'.

Because the code is eventually going to run on the Cell, we had to set up a complete environment for ourselves. We decided to use Pthreads, a library that offers typical multi-threading functionality. We will now discuss our manual implementation. Figure 5.9 shows all member variables of a FIFO and their initializations. We use a standard circular buffer plus some code to handle multithreaded access. The main buffer consists of an array of tokens, which will be floats in our case, a stored write and read position, and a boolean that keeps track of the most recent operation. We will look at the rest when we cover the multithreaded additions.

The *write_pos*, *read_pos*, and *last_action_write* variables are all used to store its state. A token is stored by writing it to the write position and read from the read position. After each operation the respective position integer is increased. If the indexes are the same, the FIFO is either empty or full. If the last operation was a write it is full and a write has to wait until at least one read is performed. If the previous operation was a read (*last_action_write* is false) and the position integers are equal, the buffer is empty and a read operation will have to wait.

The nodes on both sides of the FIFO don't synchronize their read/write operations, so without the proper measures, data corruption might occur. Luckily Pthreads, and virtually every other multithreading library, offers data structures to make sure threads access data in a safe manner. For this it uses mutexes and conditions. The former permits volatile code to only be executed by one thread at a time and the latter allows to signal other threads when a certain condition has been fulfilled.

```
for(row = 1; row <= 32; row++)
{
  for(column = 1; column <= 32; column++)
  {
    for(index = 1; index <= 32; index++)
    {
      // input code goes here...

      // call MultiplyAndAdd: the method to be implemented by programmer
      MultiplyAndAdd(C_input_value, A_input_value, B_input_value, C_outpu

      if(index < 32)
      {
        // calculated C value needs to be returned to node 5 (via FIFO1)
        write(FIFO1, C_output_value);
      }
      else
      {
        // write it to node 6 (via FIFO5)
        write(FIFO5, C_output_value);
      }
    }
  }
}
```

Figure 5.7: Code running on node 5. It is a lot more complex than node 1, because it uses three input FIFOs, of which two are buffered. The code that reads the input variables is given in figure 5.8.

```
if(index == 1)
{
  // first C-entry value needs to be read from FIFO2 (from node 4)
  read(FIFO2, C_input_value);
}
else
{
  // all other C-entries need to be read from FIFO1 (from node 5)
  read(FIFO1, C_input_value);
}

// matrix A input value needs to be read from buffer of FIFO3 (from node 1
A_input_value = FIFO3_Buffer->getFrom(row, column, index );
if (column == 32)
{
  // during last column, a row of A is read, can be removed from FIFO3 buf
  FIFO3_Buffer->releaseMem(row, column, index );
}

// matrix B input value needs to be read from buffer of FIFO4 (from node 2
B_input_value = FIFO4_Buffer->getFrom(row, column, index );
if (row == 32)
{
  // during last row, a column of B is read, can be removed from FIFO4 buf
  FIFO4_Buffer->releaseMem(row, column, index );
}
```

Figure 5.8: Code responsible for reading input variables of node 5.

```
FIFO::InitFifo()
begin
  Token buffer[buffer_size] = all empty;
  int write_pos = 0;
  int read_pos = 0;
  bool last_action_write = false;
  mutex buffer_mutex = new mutex;
  condition is_full_condition = unset;
  condition is_empty_condition = unset;
end
```

Figure 5.9: Initialization code of FIFO. Declares and initializes all local variables of a FIFO.

Every time the buffer has to be accessed, a lock on the mutex is requested. If another thread is already doing something to the data, this will make the first thread wait until this mutex will be released. After it has been acquired, a check has to be performed to see if the action can be done; in case of reading, the FIFO cannot be empty, in case of writing it cannot be full. If the test fails, the operation cannot be performed and has to be delayed until the buffer is in a correct state. A simple polling loop wouldn't do, because this would still keep the mutex acquired, therefore disallowing the other thread to work with the FIFO, bringing the system in a deadlock.

This is where the conditions come in handy. When a thread finds out that the FIFO is not in the correct state, it will set the respecting condition using *ConditionWait*. This method will atomatically unlock the mutex, set the condition and put the current thread to sleep. An other thread will now be granted access to the FIFO and will change its state, so the first thread will be able to continue. This is done after the mutex is released: after a check is performed to see if the condition is locked (*ConditionSet* will return true) *ConditionSignal* is called to wake up the original thread and place it in the execution queue.

The first thread was put asleep in the call to *ConditionWait*, so it will continue at that location in the code. Before returning control to our code, it will acquire a lock on the mutex so we will be able touch the data in the buffer knowing that it is in the correct state. This solved our synchronization problems.

5.4 A practical consideration: FIFO sizes

In most theoretical applications of KPNs, an infinite FIFO size is assumed. Needless to say, this is not very helpful in our case. Especially when one tries to store the data inside the limited Local Storage of a Cell SPE, huge buffer sizes can become fatal. The minimum required size of a FIFO of a general KPN is not decidable. We decided to find the minimum buffer size by running the

```
FIFO::WriteToken(Token token)
begin
  LockMutex(buffer_mutex);

  if (write_pos == read_pos) AND last_action_write then
    ConditionWait(is_full_condition);
  end if

  buffer[write_pos] = token;
  write_pos = (write_pos + 1) mod buffer_size;
  last_action_write = true;

  UnlockMutex(buffer_mutex);

  if ConditionSet(is_empty_condition) then
    ConditionSignal(is_empty_condition);
  end if
end
```

Figure 5.10: ‘WriteToken’. The mutex is locked, a check is performed to see if there is room in the buffer, the token is written, some bookkeeping is performed, the mutex is released, and when a reader is waiting, it is signaled that the buffer is no longer empty.

```
Token FIFO::ReadToken()
begin
  LockMutex(buffer_mutex);

  if (write_pos == read_pos) AND (not last_action_write) then
    ConditionWait(is_empty_condition);
  end if

  Token output = buffer[read_pos];
  read_pos = (read_pos + 1) MOD buffer_size;
  last_action_write = false;

  UnlockMutex(buffer_mutex);

  if ConditionSet(is_full_condition) then
    ConditionSignal(is_full_condition);
  end if

  return output;
end
```

Figure 5.11: Code called to read a token. Almost the same as the ‘WriteToken’ code, only token is read from buffer, instead of written to it. See text for complete discussion of both methods.

KNP on the PPU only while reducing the buffer size until a deadlock occurred, or until the size was acceptable. We focused on smaller problems where a buffer size of 1024 tokens turned out to be more than enough, so we used that number in all our experiments.

5.5 Our first and most simple usage of the SPEs

The method that has been described so far works perfectly on simple processors like the Intel Pentium or even on multicore-platforms where all processors communicate via shared memory and no special DMA operations are required. All code up to this point was written and debugged on a normal AMD processor. Adding the SPE functionality required some extra designing.

Figure 5.12 and 5.13 show the same simple KPN, where figure 5.12 runs on the same processor (either the AMD or the PPU) and figure 5.13 shows the KPN with nodes A and D on the SPEs.

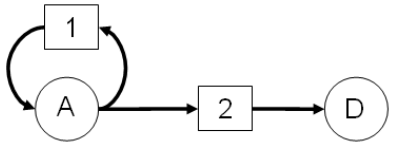

Figure 5.12: A simple KPN with two nodes and two FIFOs.

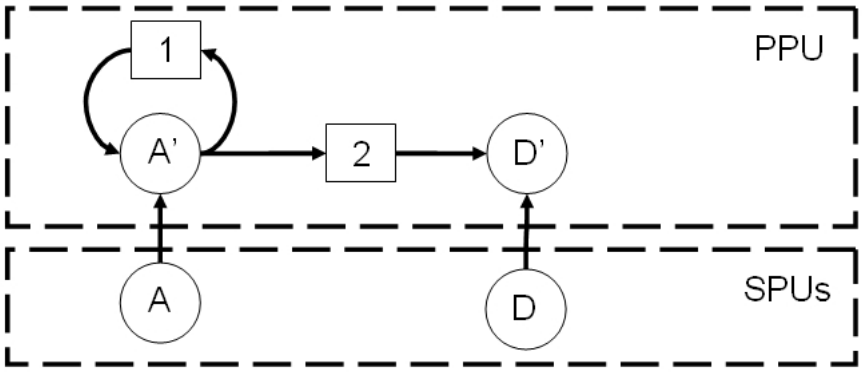

Figure 5.13: KPN of figure 5.12 put on both the PPU and on the SPEs. A' and D' act as service threads that delegate all calculations to A and D, respectively, which are located on the SPEs. A and D use their serving threads to access the FIFOs.

Figure 5.13 introduces two extra nodes on the PPU side: A' and D'. They act as service threads for their respective SPE thread. The unprimed threads in both images are the same, only the way in which they access the FIFOs differs, this will be discussed below.

As mentioned previously, the PPU and the SPEs can communicate using mail messages, this will be utilized in our communication scheme. Once a thread running on an SPE wants to perform a read or a write on one of the FIFOs (which are all located in the PPU-memory), it will write one or more outbound messages to publish this request to its serving thread on the PPU side, which, if required, will eventually write a value to the SPE's inbound mailbox.

Figure 5.14 shows what this looks like from the service thread on the PPU side, figure 5.15 shows the read, write, and stop operations from the SPE side.

The serving thread will check every iteration if there is a message to be read. If there isn't one, it will give up its control of the PPU by calling *YieldThread* (a functionality offered by Pthreads) so it will check up on the SPE's mailbox during the next scheduling round. If there is a message waiting, it will use it to find out what action the SPE wants to have performed. If it is a write operation, the SPE will send the FIFO number and the token (which we assume to be 32 bits, we always used floats so this requirement was met) so it can be written to the appropriate FIFO.

When a read is required the FIFO number is also send to the PPU. The service thread will then find the FIFO, read a token from it and send it to the SPE. And finally if a request for exit is received, the loop is exited and the thread terminates. This makes sure the thread runs no longer than absolutely necessary.

5.6 Performance of this solution

Since our main interest lays with the time required to communicate between nodes, we measure the performance of this protocol by implementing the simple KPN shown in figure 5.16; a producer-consumer network.

We used the most basic measurement of speed available: execution time. Since the Cell features a PowerPC unit, the most precise clocking functionality offered is the time base register. This register is updated a large number of times per second, so to calculate the number of elapsed second, one needs to count the number of increments and divide it by the timebase value for that particular processor. Figure 5.17 and 5.18 show the contents of the '/proc/cpuinfo' files for the PS3 and the Cell simulator respectively.

5.7 Implementing a more advanced way of inter-process communication

The previously shown method is very slow. For every token three mail messages are needed, which is too much. We decided to solve this problem by sending multiple tokens at once. This will require DMA operations that send over complete buffers and introduces extra bookkeeping.

```
enum OPERATIONS:
  OPERATION_WRITE
  OPERATION_READ
  OPERATION_EXIT
end

boolean exit := false;
SPE spe := serviced SPE;
Token token := null;

while(not exit) do
  if(MailMessageWaiting(spe)) then
    ReadFromMailbox(spe, Operation)

    switch(Operation)
      case OPERATION_WRITE do
        ReadFromMailbox(spe, FifoNum)
        ReadFromMailbox(spe, Token)
        GetFifoByNum(FifoNum).WriteToken(token)
      end
      case OPERATION_READ do
        ReadFromMailbox(spe, FifoNum)
        token := GetFifoByNum(FifoNum).ReadToken
        WriteToMailbox(spe, token);
      end
      case OPERATION_EXIT do
        exit := TRUE
      end
    end
  else
    YieldThread;
  end
end
```

Figure 5.14: Service thread on PPU side. This thread runs while the thread on the SPE side runs. It monitors the outbox of its designated thread. See the text for a complete discussion.

```
WriteToken(Fifo fifo, Token OutToken) :
  WriteToOutbox(OPERATION_WRITE);
  WriteToOutbox(fifo.GetNumber);
  WriteToOutbox(OutToken);
end

ReadToken(Fifo fifo, Token & InToken) :
  WriteToOutbox(OPERATION_READ);
  WriteToOutbox(fifo.GetNumber);
  ReadFromInbox(InToken);
end

Exit() :
  WriteToOutbox(OPERATION_EXIT);
end
```

Figure 5.15: Implementation of the three methods used by the code running on an SPE. All actions are implemented by sending and receiving the required mail messages.

Figure 5.16: A producer-consumer KPN. This network is used to measure the performance of the two protocols developed during this project. By moving the sending and receiving nodes to different Cell units the speed of all communications can be determined.

This will be dealt with below.

Figure 5.20 shows the operations performed when filling a buffer and sending it using DMA. For now we only consider a buffer size of four tokens, but later on we will experiment with a larger buffer. In our solution the buffer is filled by copying tokens to a linear buffer. After then it is copied to a memory space of the same size in the memory of the one were the reading node is located. When the DMA is competed, a message is send to the writing node, so it can consider the buffer to be writable again. The writing node has to wait for the DMA completion before writing, otherwise data consistency cannot be guaranteed.

All DMA operations are initiated by the SPE-side of the FIFO, this is because, according to the official IBM Cell manual, the performance is better and also because it allows us in the future to set up more concurrent DMA operations: when multiple connections need to be served, multiple DMA tags (description IDs) could be used.

```
processor : 0
cpu : Cell Broadband Engine, altivec supported
clock : 3192.000000MHz
revision : 5.1 (pvr 0070 0501)

processor : 1
cpu : Cell Broadband Engine, altivec supported
clock : 3192.000000MHz
revision : 5.1 (pvr 0070 0501)

timebase : 79800000
platform : PS3
```

Figure 5.17: '/proc/cpuinfo' for the PlayStation 3. The two hardware threads are shown as separate processors (values given: index, description, clock speed and revision). Important to us: the 'timebase' value, it gives the number of increments of the 'time base' register per second.

```
processor : 0
cpu : Cell Broadband Engine, altivec supported
clock : 3200.000000MHz
revision : 5.0 (pvr 0070 0500)

processor : 1
cpu : Cell Broadband Engine, altivec supported
clock : 3200.000000MHz
revision : 5.0 (pvr 0070 0500)

timebase : 25000000
platform : Cell
machine : CHRP IBM,CPBW-Mambo, Simulated-System
```

Figure 5.18: Contents of 'cpuinfo' for the Cell simulator. It has a smaller timebase value (to lower the strain on the simulator), but does run on the theoretical speed of 3.2 GHz.

	Number of tokens	Time base increments	Time (s)	Tokens per sec
PPU to PPU	2^{10}	25567	0.00032	$3.1 \cdot 10^6$
	2^{20}	204161972	2.56	$4.0 \cdot 10^5$
	2^{27}	22226295746	278.53	$4.8 \cdot 10^5$
SPE to PPU	2^{10}	2563002	0.032	$3.2 \cdot 10^4$
	2^{20}	2972280622	37.24	$2.8 \cdot 10^4$
	2^{27}	380493349115	4768.08	$2.8 \cdot 10^4$
PPU to SPE	2^{10}	1635467	0.020	$4.9 \cdot 10^4$
	2^{20}	2824044831	35.39	$2.9 \cdot 10^4$
	2^{27}	364814712356	4571.61	$2.9 \cdot 10^4$
SPE to SPE	2^{10}	2807189	0.035	$2.9 \cdot 10^4$
	2^{20}	2973432971	37.26	$2.8 \cdot 10^4$
	2^{27}	391067207313	4900.59	$2.7 \cdot 10^4$
PPU to self	2^{10}	63639	0.00079	$1.2 \cdot 10^6$
	2^{20}	51389370	0.64	$1.6 \cdot 10^6$
	2^{27}	6471958374	81.1	$1.6 \cdot 10^6$
SPE to self	2^{10}	6242437	0.078	$1.3 \cdot 10^4$
	2^{20}	4784318814	59.95	$1.7 \cdot 10^4$
	2^{27}	450529293916	5645.73	$2.3 \cdot 10^4$

Figure 5.19: Our first, service-thread-based, solution's performance. All connections were tested for 2^{10} (1024), 2^{20} ($1024 \cdot 1024$), and 2^{27} ($128 \cdot 1024 \cdot 1024$) tokens and the required execution time was used to calculate the effective number of tokens per second.

If a connection is between two SPEs, the DMA is initiated by the reading party. It is partly a matter of definition, but also assures every SPE only accesses any other SPE's in read-only mode, this is in general a good strategy. It is not so much an issue when performing SPE to PPU DMA operations, because the SPE will not be reading or writing into the PPU's local storage (cache), but accesses the main memory instead. The PPU's internals will make sure to flush its cache beforehand.

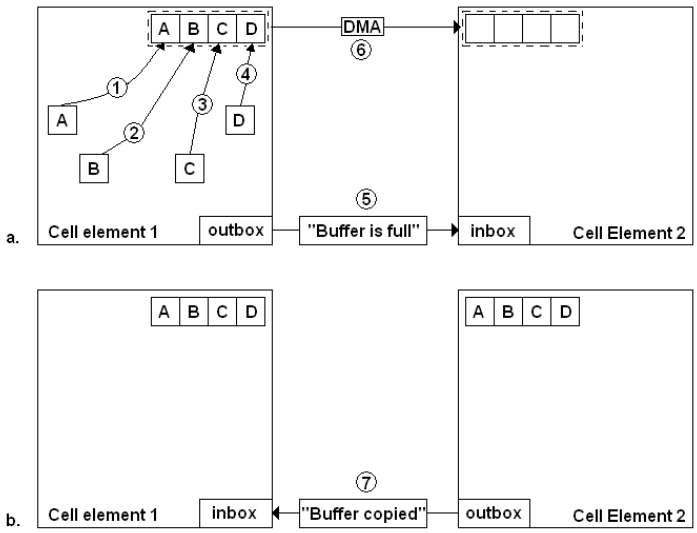

Figure 5.20: Usage of DMA to offer a FIFO connection between two elements of the Cell processor. Steps 1 through 4 copy values to the buffer in element 1. Step 5 consists of sending a message to the reader to tell it the buffer is full, so it can start the DMA operation (step 6). After this is finished, a message is sent back to tell the buffer has been copied and can again be written to.

For this to work, the connection needs to be set up correctly. One obvious requirement is to have the party that is starting the DMA to know the location of the buffer. Also some information about the buffer has to be stored so it can be linked up with a FIFO number.

To make this system work, we had to give every FIFO not just a global number but also a local number. A FIFO is accessed in a local table given the number of the element on its other side (which is either a PPU thread or an SPE) and the index of the FIFO for that particular element pair.

Figure 5.21 shows a useful KPN which uses all connection types. We will use them to explain how their respective connection is dealt with using all shown methods:

- PPU to PPU: this is still accomplished using the FIFOs set up for multi-thread access.

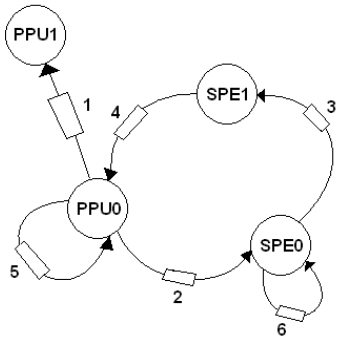

Figure 5.21: A 4-node KPN running on the PPU and two SPEs. All connection types are featured; 1: PPU to PPU, 2: PPU to SPE, 3: SPE to SPE, 4: SPE to PPU, 5: PPU self loop, and 6: SPE self loop. We needed to deal with all of them separately.

- PPU to SPE, SPE to SPE and SPE to PPU: for this the newly introduced buffered communication is used.
- PPU self loop and SPE self loop: use a simple circular buffer (comparable to the PPU to PPU FIFOs, without the mutexes).

Figure 5.22 shows the the performance of this new method. Figure 5.23 shows what we already expected when looking at the numbers: method 2 is much faster.

We will now explain the measured results. The fact that method 2 is just as fast on PPU to PPU communications as method 1 is because they use the same FIFO code. The speed-up for the SPE to PPU, PPU to SPE and SPE to SPE operations is because of the almost complete removal of the mail messaging, only once every 1024 tokens a message needs to be sent to tell the reading side to copy the buffer.

The self loops are faster because they are no longer implemented with a mutexed FIFO, but via a simple looped buffer, this works because we know that its accesses will always be done by the same thread. Especially the SPE self loops are much faster. No longer a token needs to be send to the PPU and read back (which requires 6 mail messages for every token), it is just written to and read from local storage.

It now becomes very apperent that the PPU to PPU communication is very slow in comparison to the other types and it is therefore smart to try to move all nodes to the SPEs.

5.8 Flushing operations

The comparison between the two methods might be considered a bit unfair. Until now we left one detail out of our discussion, namely the requirement for the flushing of tokens. This is were

	Number of tokens	Time base increments	Time (s)	Tokens per sec
PPU to PPU	2^{10}	158535	0.0019	$5.1 \cdot 10^5$
	2^{20}	222215460	2.78	$3.7 \cdot 10^5$
	2^{27}	27385823967	343.18	$3.9 \cdot 10^5$
SPE to PPU	2^{10}	19512	0.00024	$4.1 \cdot 10^6$
	2^{20}	4958693	0.062	$1.6 \cdot 10^7$
	2^{27}	632744043	7.92	$1.6 \cdot 10^7$
PPU to SPE	2^{10}	14616	0.00018	$5.5 \cdot 10^6$
	2^{20}	4621259	0.057	$1.8 \cdot 10^7$
	2^{27}	589762084	7.39	$1.8 \cdot 10^7$
SPE to SPE	2^{10}	62098	0.00077	$1.3 \cdot 10^6$
	2^{20}	13278517	0.16	$6.3 \cdot 10^6$
	2^{27}	693700814	8.69	$1.5 \cdot 10^7$
PPU to self	2^{10}	47008	0.00058	$1.7 \cdot 10^6$
	2^{20}	2419728	0.030	$3.4 \cdot 10^7$
	2^{27}	303528609	3.8	$3.5 \cdot 10^7$
SPE to self	2^{10}	4778	0.000059	$1.7 \cdot 10^7$
	2^{20}	2836617	0.035	$2.9 \cdot 10^7$
	2^{27}	362437027	4.54	$2.9 \cdot 10^7$

Figure 5.22: Performance for our second implementation, which is DMA based. The buffer size is 1024 tokens.

	Method 1 base incs.	Method 2 base incs.	Speed-up
PPU to PPU	22226295746	27385823967	0.81
SPE to PPU	380493349115	632744043	601
PPU to SPE	364814712356	589762084	619
SPE to SPE	391067207313	693700814	564
PPU to self	6471958374	303528609	21
SPE to self	450529293916	362437027	1243

Figure 5.23: Comparison of the two discussed methods. Method 1 is service-thread-based, method 2 is DMA based. The third column gives the speed-up of method 2 over method 1. Only the PPU to PPU method has comparable performance, because this functionality is implemented the same.

a buffer is not yet full, yet needs to be read by the reading party.

Flushing operations are needed on a lot of places when writing computer programs. When writing a program that needs to log errors to a file, one needs to perform a flush operation after every experienced error, because a crash of the program will lose the contents of the buffered data sent to the file. Flushing this buffer regularly (at the end of a line, for example) makes sure all data is written to the file and a sudden crash will not cause a loss of this information.

When implementing a KPN, a flush operation is sometimes needed. A hypothetical example is shown in figure 5.24. This very simple KPN can cause a deadlock when there are no flush operations performed. Node 2 waits until it can read a token from node 1. Node 1 does write a token to node 2, but because the buffer is not yet full, it will not be copied to Node 2. Node 1 then waits for node 2 to write a token, which will never happen. With all nodes in a blocking state, a deadlock has occurred.

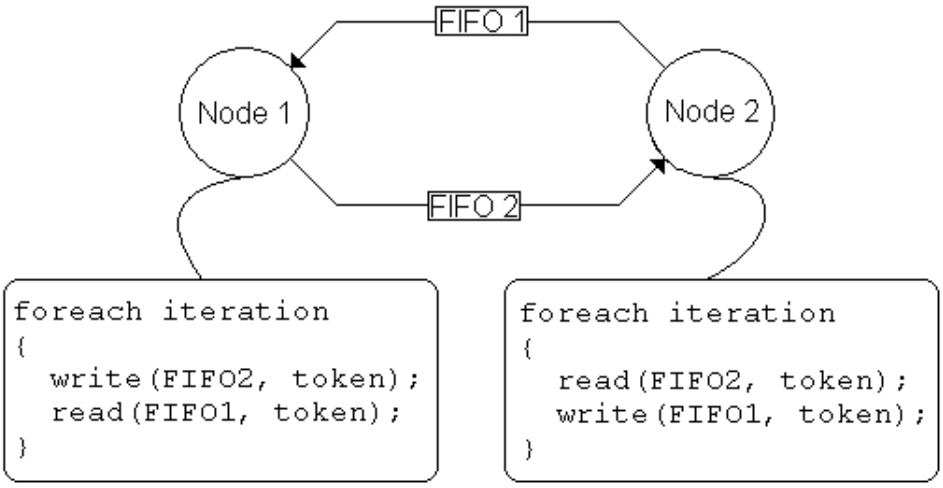

Figure 5.24: A simple KPN terminating in a deadlock situation if no flushing is performed. After node 1 has written a token, but doesn't flush FIFO 2, both nodes are blocked in read state.

If the FIFOs would have been flushed after every write operation, then no deadlock would occur. This was never a problem in the old method (where every token could be accessed directly after it was written to the FIFO), but is now introduced because of our buffered DMA-based approach.

If we perform the same measurements as we did previously for the other methods (see figure 5.25), we see that we still manage to achieve a speed-up, but we lose considerably compared to the fully buffered DMA method. This makes sense, because every token needs a few mail messages and every DMA operation is just as large as if the complete buffer was filled.

Figure 5.26 shows that reducing the buffer size doesn't improve the performance of the communication. This shows us that the performance of the communication is independent from the

	Base clock incs.	Method 1 speed-up	Method 2 slow-down
PPU to SPE	123507888768	2.95	209
SPE to PPU	127623656934	2.98	201
SPE to SPE	244037217562	1.60	351

Figure 5.25: Effect of continuously flushing on performance of communicating 2^{27} tokens. Comparison is made to method 1 (service-threads) and method 2 (fully buffered DMA). The used buffer size is 1024 tokens.

	Base clock incs.	Tokens per second	Speed-up over 1024
PPU to SPE	122923777485	$8.7 \cdot 10^4$	1.005
SPE to PPU	127362109930	$8.4 \cdot 10^4$	1.002
SPE to SPE	246506855335	$4.3 \cdot 10^4$	0.990

Figure 5.26: Performance of method 2 when continuously flushing the DMA buffers compared to method 1 and the original method 2. The buffer size is reduced to 4 tokens, to see what it does to the required time to communicate compared to the 1024 buffer size.

buffer size, most likely caused by the way the DMA is implemented: a relatively high start-up cost, but almost no extra cost when increasing buffer size. This then shows us that we don't have to expect an increase in performance when we reduce the buffer size.

Chapter 6

Conclusion

We were able to achieve a speed-up of 6.5 by implementing SART on the Cell. We also have implemented two different communication schemes to offer the required multi-threaded environment so the code generated by COMPAAN works on the STI Cell. This offered us the possibility to measure its performance and get a feeling of how well these solutions did.

Part of the project was to write a back-end for COMPAAN so the produced KPN is used to produce the C code to communicate with the manually written Cell communication library. This reduced the long-term development time, because we now only have to make some changes in the KPN (by editing the matlab file) and regenerate the C-files, compile and measure the performance.

This project has helped to improve the performance of SART and also to gain a better understanding on how to parallelize code for the STI Cell.

Appendix A

Description of files generated by CellCC

The back-end of COMPAAN responsible for generating the C/C++ code for the Cell, CellCC, writes to a number of files. This appendix covers the meaning of these files and the variables declared in them.

A.1 types.h

This file contains a number of global variables, that are used by the PPU and SPE code:

- *g_iSPECount*: Number of SPE threads, needs to be exactly right.
- *g_iPPUThreadCount*: Number of PPU threads, can be larger than the actual number of threads.
- *g_iMaxPPUSPEBuffers*: Maximum number of connections between a PPU and an SPE thread. Can be larger than actually required.
- *g_iMaxSPESPEBuffers*: Maximum number of connections between two different SPE threads.

A.2 ppu_buffers.h

Contains all variables used only by the PPU side. Buffers are statically allocated. They need to be quad word aligned (done in gcc by using the '__attribute__ ((aligned (16)))' attribute) and their size is required to be a multiple of 16 bytes (128 bits). We have stored floats during the course of this project, which meant the number of elements of the buffers needed to be a multiple of 4. It is important that the SPE and the PPU consider their shared buffers to be of equal size, we did this by defining their size in types.h and perform only their static allocation in ppu_buffers.h.

- *g_pSPEProgramHandles*: an array that contains all spe_program_handle_t pointers to the embedded binaries that need to run on the SPEs.
- *g_pBufferLocations*: contains all void pointers to the statically allocated buffers.
- *g_iBufferSizes*: contains the number of tokens for all buffers.
- *g_pBufferDescs*: stores a descriptor string for every buffer. Is only used when debugging.
- *g_pBufferNames*: stores buffer names (very short string to relate it their FIFO number).
- *g_iMaxPPUThreadSelfLoops*: the maximum number of self loops in a PPU thread. Can be larger than required, not smaller.
- *g_iSelfLoopCounts*: array that stores the number of self loops for every thread, needs to be precise.
- *g_iMaxPPUThreadFifos*: stores the maximum number of FIFOs between two sperate threads, can be larger than the actual value, not smaller.
- *g_PPUFifos*: stores the FIFO start and end threads for all PPU to PPU communications. Every entry is an array of two unsigned ints: the start thread index and the end thread index.
- *g_iPPUToSPEBlockingType*: stores the blocking type used when sending messages to SPEs. It needs to be *SPE_MBOX_ALL_BLOCKING* so all messages are received correctly by the SPEs. This value means the mail message sending call only returns when all messages could be written.

A.3 PPU Makefile

Thanks to a well designed makefile-system offered by IBM, building a executable for the Cell is very easy. Only a few fields need to be filled out in the Makefiles created by the programmer.

- *PROGRAM_ppu64*: to this variable the name of the output file needs to be assigned.
- *IMPORTS*: needs to point to the a-files produced in the spu-directory (more about those below) and needs to include ‘-lspe2’.

To make it work, this file also needs to include the make.footer file shipped with the SDK.

A.4 spu_buffers.h

Just as with 'ppu_buffers.h', this file contains the static allocation of all buffers used by the read/write code, only now for the SPEs. All buffer allocations or all SPEs are declared, yet when it is compiled, every SPE uses only a part of it. This is done by defining COMPILING_SPE-preprocessor directives. This makes sure the 'spu_readwrite.h' file can included the file without worrying about its SPE index.

The following variables are required:

- *g_iSPEIndex*: every SPE has its own index, this is used when printing debug messages to the console. It starts at zero.
- *g_pBufferLocations*: stores all addresses of the statically allocated buffers.
- *g_iBufferSizes*: stores all buffer sizes.
- *g_pBufferNames*: stores the names of all allocated buffers.
- *g_pBufferDescs*: stores the descriptions of the buffers.
- *g_iInPortsToPPU*: stores the number of input ports served to every PPU thread.
- *g_iOutPortsToPPU*: stores the number of output ports to every thread.
- *g_iInPortsToSPE*: stores the total number of input ports served from other SPEs.
- *g_iOutPortsToSPE*: stores the total number of output ports served to other SPEs.
- *g_iInportSPEs*: stores an array where every value v at index i equals the served SPE v by input port i.
- *g_iOutportSPEs*: the same as the previous variable, but then for outports.
- *g_iInportCountForSPE*: array of integers storing the number of input ports for all other SPEs.
- *g_iSelfLoopCount*: number of self loops used by this SPE.

A.5 SPE Makefile

Just as for the PPU part, a makefile needs to be written for the SPE code. Only two variables need to be defined:

- *PROGRAMS_spu*: lists all SPE programs made visible to the PPU code. These will be listed in the *g_pSPEProgramHandles* variable in the 'ppu_buffer.h' file.
- *LIBRARY_embed64*: contains all a-files belonging to the program handles (in the same order). These files will be include in the ppu-makefile.

A.6 main.cc

The entry point of the application is stored inside the ppu directory. Besides all thread functions, the main function itself is also required. This function is used to call all initialization and finalization code and to start the PPU and SPE threads. The following things need to be done in the given order:

- *InitAllPPUThreadData is called*, this call initializes all data stored to manage the PPU and SPE threads;
- *InitSPEThreads()* is called next. It will setup and start all SPE threads;
- *InitPPUBuffers()* is finally called setup the PPU side buffers.

Then the PPU threads need to be started manually, followed by a call to join them. This is al done with calls the methods *pthread_create* and *pthread_join* offered by Pthreads. Then *WaitForSPEsFinished* is called and a special line of assembly code is inserted to assure all SPE threads are finished (the call is 'sync', offered by the Cell assembler). Then *JoinAllSPEThreadFunctions* is called to clean up all SPE running threads.

A.7 SPEx.cpp

The code running on the SPEs is stored in the SPEx.cpp files, with x a number between 1 and 6. As explained in the section about spu_buffers.h, every time code is compiled for an SPE it needs to define a preprocessor directive called 'COMPILING_SPEx'. There are two more requirements when writing a C++ file for an SPE:

- Before anything else *InitSPEBuffers* needs to be called, this call initializes the local buffers by communicating with the PPU (via mailmessages).
- When the SPE is done, *SendExitMessage* should be invoked. This call tells the PPU it is done. The PPU uses this to keep track of which SPEs are still running (this then is used to continue the forwarding of mail messages between SPEs).

Appendix B

Usage of code base

A lot of code has been written during this project. This section explains what code is most important and also how it is used.

B.1 sart/

This directory contains a lot of cpp files that all implement a different part of this project:

- 'basicsart.cpp': original implementation of SART done by Joost Batenburg.
- 'projectimage.cpp': projects an image (stored in a text file) using a matrix (matrix W).

This directory also contains a script called 'compile.sh'. Running this file will take care of all steps needed to generate the image file, produce the matrix file and compiling and running the SART algorithm. The following variables can be set in this script:

- *MATRIX*: name of matrix file (can be any file name)
- *SIMPLESPHERE*: name of generated image file, most likely simplesphere.txt, an image containing a number of opaque circles.
- *PROJECTION*: name of the output file to store the projection.
- *OUTIMG*: file name of where to store the restored image (this can later be used to calculate the error).
- *TOMSETTING*: file name of settings file used by code that generates the matrix file.
- *ITERCOUNT*: number of SART iterations.
- *ANGLECOUNT*: number of projection directions.

- *IMAGESIZE*: number of pixels at one size of the image. The actual number of pixels in the image will actually be *IMAGESIZE* times *IMAGESIZE*.
- *PTDETAIL*: level of detail when projection is performed. A value of 100 means that the width of the strips is the same width as a pixel. A smaller value will increase the width and lower the number of strips, therefore reducing the number strips and execution time of the algorithm.

B.2 sart/writematrix/

This directory contains the code that produces the matrix file used by the projection and SART code. Run it without arguments to see its required usage.

B.3 sart/bmp2img/

This directory contains functionality to convert a bitmap image to an image text file used by SART and the other way around. Run this program without arguments to see how it should be used.

B.4 sart/componized/sanlp_pthreads/

This directory contains the output generated by COMPAAN of SART (for yapi) with a manual implementation of the multi thread environment based on Pthreads. The following files are required to make code for yapi work with Pthreads: ‘fifo.h’, ‘id.h’, ‘process.h’, ‘RTE.h’, and ‘yapi.h’.

B.5 sart/cell/

This directory contains a number of projects that were used to create the two communication methods. The following directories are most interesting:

- ‘oldstyle_comm_measure’: used to measure the speed of method 1 (service-thread-based).
- ‘comm_measure’: used to measure the speed of DMA buffered communication.
- ‘comm_measure_flushed’: is used to measure the speed of the DMA buffered method, when it is constantly flushed.
- ‘sanlp_spu’: implementation of SART using method 1.

- ‘sart_cell_all’: manual implementation of SART, gives the speed-up of 6.5.

Bibliography

[1] Ana Lucia Varbanescu, Doctoral Track within the Parallel and Distributed Group of the TU Delft, Faculty of Engineering, Mathematics and Computer Science (EWI): http://www.st.ewi.tudelft.nl/∼varbanescu/

[2] Avinash C. Kak and Malcolm Slaney. “Principles of Computerized Tomographic Imaging”. Society of Industial and Applied Mathematics. Online available at: http://cobweb.ecn.purdue.edu/∼malcolm/pct/.

[3] International Business Machines (IBM). “STI cell processor, New Disclosures to Jumpstart Creation of Cell-based Applications Beyond Gaming”: http://www-304.ibm.com/jct03004c/businesscenter/venturedevelopment/us/en/feature-article/gcl_xmlid/8649/nav_id/emerging.

[4] International Business Machines (IBM). “The Cell project at IBM Research”: http://www.research.ibm.com/cell/.

[5] Sony Computer Entertainment Inc. “PlayStation Global”: http://playstation.com/.

[6] International Business Machines (IBM). “IBM Full-System Simulator for the Cell Broadband Engine Processor”: http://www.alphaworks.ibm.com/tech/cellsystemsim.

[7] Leiden Institute for Advanced Computer Science (LIACS). “Compilation of Matlab to Process Networks (Compaan)”: http://www.liacs.nl/ cserc/compaan/.