University of Alberta

Low Power Real Time 3D Rendering on an Embedded SIMD Processor

by

Jeffery Scott Mrochuk

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Electrical and Computer Engineering

Edmonton, Alberta
Spring 2008

Your file Votre référence
ISBN: 978-0-494-45859-4
Our file Notre référence
ISBN: 978-0-494-45859-4

Abstract

Graphical rendering is implemented on a Single Instruction Multiple Data (SIMD) processor array using two styles of parallelism. The SIMD processor is evaluated as a platform for real time 3D rendering in a low power mobile environment, using the SIMD Array Processor and ARM922T™ CPU within the Atsana Semiconductor J2210 Media Processor. The first algorithm is tile based, and treats the Array Processor as an intelligent frame buffer. The second algorithm uses each processor to run a simple shader algorithm on one or more pixels. The SIMD tile algorithm shows a 10.5 times performance increase and is 8.4 times more energy efficient than the ARM on the simplest tests, but performance degrades in complex real world cases. The pixel algorithm shows a SIMD performance which exceeds 5 times the sequential algorithm and is 7.7 times more energy efficient than the ARM, but exposes memory bandwidth issues in the J2210.

Table of Contents

1 Introduction **1**
1.1 Overview 2
1.2 Thesis Organization 3

2 Background **5**
2.1 SIMD Computers 5
2.1.1 SIMD Hardware for Experimentation 7
2.1.2 Prior Use of SIMD and Graphics 8
2.1.3 SIMD Low Power Applications 9
2.2 Processor in Memory Architectures 11
2.3 Development Platform 12
2.4 3D Rendering Process Overview 16
2.4.1 Implementation of the 3D Rendering Process 18
2.5 Low Power Embedded Rendering Applications 19
2.6 Summary 21

3 Data Parallelism in 3D Rendering **23**
3.1 Vertex Processing Data Parallelism 24
3.1.1 Vertex Transformation Stage 25
3.1.2 Perspective Transformation Stage 26
3.1.2 Lighting Stage 27
3.1.3 Vertex Acceleration on the J2210 28
3.2 Pixel Processing Data Parallelism 29
3.2.1 Attribute Interpolation 29
3.2.2 Tile Based Parallelism 31
3.2.3 Pixel Based Parallelism 34
3.3 Summary 37

4 Tile Based Rendering Approach **38**
4.1 Host/Array Communication Implementation 42
4.2 Triangle Drawing Implementation 45
4.3 Gouraud Shading Implementation 50
4.4 Depth Calculation and Evaluation 53
4.5 Texture Coordinate Calculation 55
4.6 Performance 58
4.7 Power and Energy 60
4.8 Memory 64
4.9 Summary 65

5 Pixel Based Rendering Approach **67**
5.1 Host/Array Division of Work 69
5.2 Host/Array Communication Implementation 71

5.3 Triangle Drawing 74
5.4 Gouraud Shading Implementation 76
5.5 Depth(Z) Calculation and Evaluation 79
5.6 Texture Coordinate Calculation 80
5.7 Performance 82
5.8 Power and Energy 87
5.9 Memory 91
5.10 Summary 92

6 System Level Communications and Efficiency 94
6.1 System Level Parallelism 95
6.2 Data Movement 96
6.2.2 Host processor Read/Write 96
6.2.2 J2210 DMA 97
6.3 Potential Hardware Improvements for 3D Rendering 98
6.3.1 System/Array DMA 98
6.3.2 Display output DMA 99
6.3.3 Optimal CU and memory configurations 100
6.4 Summary 100

7 Conclusions 101
7.1 Future Research Directions 102

Bibliography 105

List of Figures

2.1: A Basic SIMD Architecture 6
2.2: Atsana J2210 Customer Evaluation Board 13
2.3: J2210 High Level Block Diagram 15
2.4: 3D Rendering Process Overview 16
3.1: 3D Rendering Process Breakdown 24
3.2: Pixel layout in the array as it forms the complete frame buffer 33
4.1: Division of Rendering work 39
4.2: Pixel layout in the array as it forms the complete frame buffer 40
4.3: Memory layout in the CUs of the memory array 43
4.4: Definition of the two orientations of triangles dealt with by the algorithm 46
4.5: Example line intersections forming the write enable array to draw the triangle 47
4.6: SIMD array painting a white triangle with three intersecting lines. 49
4.7: Two triangles drawn and Gouraud shaded simultaneously by different CUs 52
4.8: The triangle figure above, this time with another triangle masking out a portion 54
4.9: Four Cubes drawn using the depth test and Gouraud shading algorithm 55
4.10: A black and white stripe texture modulated with the Gouraud shading 58
4.11: The Newell teapot rendered by the SIMD array with texturing disabled 60
5.1: Parallel Execution Comparison 69
5.2: Processor division of work 70
5.3: Scanline processing algorithm implemented in the ARM 75
5.4: Division of work for single pixel per CU. 43 CUs are used to fill the triangle 77
5.5: Division of work for four adjacent pixels per CU, 14 CUs are used 78
5.6: A triangle obscures the first triangle, reducing the number of pixels sent to the AP 80
5.7: Processing time comparison for varying patch sizes 84
5.8: Processing time comparison for varying patch sizes including the 4 pixel array processor neighbour calculation 85
5.9: Energy comparison for varying patch sizes including the 4 pixel array processor neighbour calculation 91

List of Tables

3.1: Pseudo code for pixel processing 30
3.2: Pseudo code for CU interpolation 34
3.3: Breakdown of addition operations on ARM and AP 35
3.4: Breakdown of multiplication operations on ARM and AP 36
4.1: Code for packing and writing short integers to the Array Processor 44
4.2: Calculations for fixed point interpolation coefficients 50
4.3: Pseudo code for interpolation algorithm 51
4.4: Pseudo code for depth test 54
4.5: Pseudo code for texture calculation 56
4.6: Clock cycle breakdown of algorithm stages 58
4.7: Hardware vs Simulator performance measurement 59
4.8: Current consumption of the test cases 63
4.9: Power consumption broken down by hardware 63
4.10: Energy consumption 64
4.11: Memory Usage 65
5.1: Pseudo code for triangle inclusion test 71
5.2: Pseudo code for pixel processing 72
5.3: Algorithms in millions of operations per second 82
5.4: Algorithms in millions of operations per second, throughput 83
5.5: Number of blocks required to cover an orthogonal isosceles triangle of varying size 86
5.6: Number of cycles to calculate all the pixels in orthogonal isosceles triangle, based on varying block sizes 86
5.7: Number of blocks required to cover a line of varying size 87
5.8: Number of cycles to calculate all the pixels in a line, based on varying block sizes 87
5.9: Current consumption of test cases 89
5.10: Power consumption broken down by hardware 89
5.11: Energy consumption of 192,000 single pixel calculations 90
6.1: Speed of read and write operations from ARM to AP 96

Nomenclature

List of Acronyms

AC Array Controller

ALU Arithmetic Logic Unit

AP Array Processor

API Applications Programming Interface

CEB Customer Evaluation Board

CIU CMEM Interface Unit

CMEM Computational Memory

CPU Central Processing Unit

CU Computational Unit

DMA Direct Memory Access

DSP digital signal processor

FPGA Field Programmable Gate Array

GPU Graphics Processing Unit

IC Integrated Circuit

PE processing element

PiM Processors-In-Memory

SEL SIMD Engine Language

SIMD Single Instruction Stream, Multiple Data Streams

SISD Single Instruction Stream, Single Data Stream

SoC System-on-Chip

Chapter 1
Introduction

Portable technology is becoming more prevalent every day. Most people are never without some sort of portable phone, music player or organizer. The requirements on these systems are conflicting, with the desire to be feature rich and powerful while maintaining a long battery life time. Originally primitive devices with monochrome displays have evolved to elegant devices with robust user interfaces and full colour displays. The demand on these devices to support both traditional productivity applications as well as entertainment has added a new challenge to the designer to push processing power up while keeping energy consumption down. This thesis investigates the computationally demanding application of real time 3D rendering on a low power mobile embedded platform such as a cellular phone.

1.1 Overview

A low power Single Instruction Multiple Data (SIMD) hardware architecture is used to study the effectiveness of 3D rendering algorithms on a massively parallel, yet low power, platform. A SIMD architecture operates using a single instruction stream sent to multiple processors operating on their own individual data sets. While sequential processors which operate on a single data set can be described as Single Instruction Single Data (SISD), SIMD machines operate simultaneously on multiple data sets, and is therefore described as Single Instruction Multiple Data. Hardware requirements are intensive for 3D rendering applications and require alternative methods to SISD computation, due to the large amounts of individual computations that need be performed on large data sets.

SIMD systems rely on potential parallelism inherent in the algorithms that are operating on the large data sets. Often the same operation is performed again and again on a data set in multimedia applications, such as accumulation by variable. When the data set is not dependent on previously calculated results, it is possible to do these operations in parallel. In 3D rendering every object must be filled with pixels in order to fill the screen's display with a 2D representation of the 3D scene. In a pixel processing loop each pixel may have a colour filter applied to the existing colour of the surface, for example. Thousands or millions of pixels may have the same operation performed using a logical or arithmetic

instruction. The ability to perform these operations in parallel can increase the throughput of the system in direct proportion to the number of processing elements (PEs). 3D rendering has large amounts of parallelism available at the vertex and pixel level.

The power savings are attained on this platform by using a Processor-In-Memory (PiM) architecture. PiMs have traditionally yielded low energy consumption while performing computation, if the application's requirements are suitable to the architecture, due to more efficient use of data.

This dissertation shows that while certain aspects of the rendering application are both suitable and proficient in a SIMD PiM platform, there are underlying communication flaws in the architecture which prevent it from becoming a fully effective solution.

1.2 Thesis Organization

Seven chapters including this introductory chapter comprise this thesis.

Chapter 2 is a summary of the background information required for this research. It provides a summary of past SIMD processors, including some research done with the 3D rendering application on SIMD architectures. It also provides an overview of the J2210 development platform used in this study.

Chapter 3 provides the possible approaches for accelerating rendering algorithms using the platform. It provides solutions for parallelizing the major stages of the 3D rendering process, and motivates the solutions which were

chosen for this experiment. Chapter 3 also provides the description of the software vertex processing created for this project.

Chapter 4 describes the implementation and results of the first of two SIMD solutions for the 3D implementation on the J2210. This solution parallelizes the algorithm over object space, filling the triangles with the array processor.

Chapter 5 describes the implementation and results for the second pixel solution. This solution parallelizes the algorithm over pixel space, with each processing element in the array processor calculating independent pixel sets.

Chapter 6 provides some alternative choices for future implementations of the hardware in order to minimize the issues seen in developing the prior two acceleration solutions on the J2210 hardware.

Chapter 7 draws the conclusions and provides a summary of the main contributions in this thesis. It also touches on some areas of interest for future research.

Chapter 2
Background

This project combines several computer architecture concepts. The primary focus is the inherent parallelism of 3D rendering, when applied to a Single Instruction Multiple Data (SIMD) computer architecture. The particular SIMD platform used is a Processor in Memory (PiM) architecture, which provides extremely high memory bandwidth to the processors. The goal is to put these three topics together for fast low power rendering in an embedded environment. Specifically we explore the use of SIMD for efficient parallel computations, the concept of PiM for lower power consumption and high bandwidth, and the application of a 3D rendering engine to this architecture.

2.1 SIMD Computers

Early applications of SIMD were in large scale supercomputers in the 1970s and 1980s. Generally designed for scientific calculations, processors such as the ILLIAC IV, the Massively-Parallel Processor and the Connection Machines 1&2,

were physically large and powerful [1]. They used arrays of processors to perform parallel calculations on large data sets using a dedicated ALU and some amount of private memory per Processing Element (PE). Each is a two dimensional grid of processors featuring some form of communication interconnect for data movement. Generally these arrays have some sort of interface that communicates with a more traditional SISD processor to provide a constant stream of instructions. As well, the processing elements have some sort of interconnect network for communication as shown in Figure 2.1.

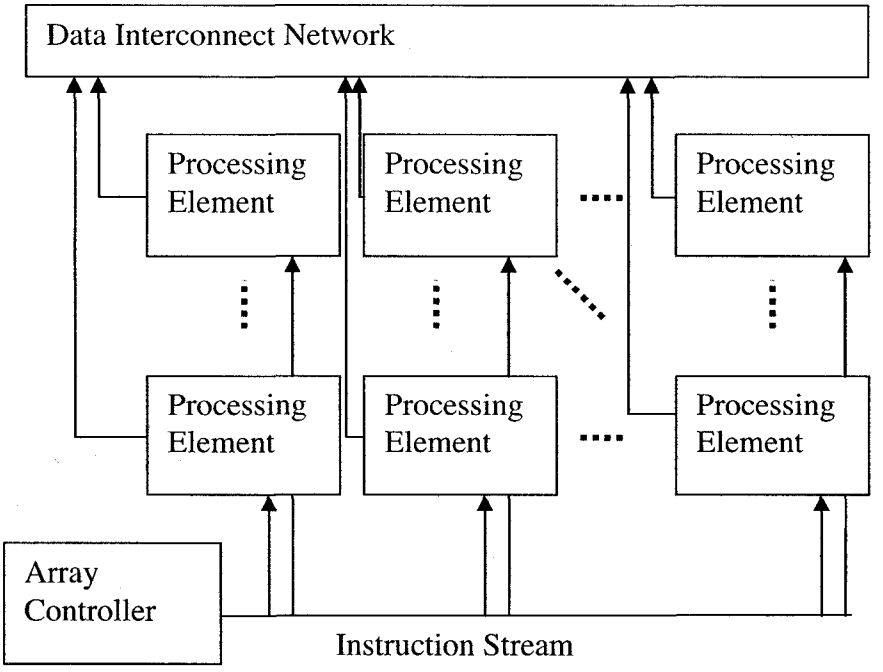

Figure 2.1: A Basic SIMD Architecture

While large scale SIMD, with dozens of processors, has not become popular in personal computer CPU technology, the SIMD design has been applied to smaller scale arithmetic processing units. Intel's MMX instruction set allows simultaneous parallel integer operations through a specialized Arithmetic Logic Unit (ALU) [2]. It uses the architecture's existing 64 bit registers as an array of two, four or eight integer values for processing simultaneously with the same instruction. Later Intel added the SSE series of instruction sets which has its own dedicated 128 bit registers and added floating point support [3]. Hardware and instruction set extensions like MMX and SSE have been the limit of SIMD architecture in main system processors in recent years, but have provided significant performance benefits in suitable applications.

2.1.1 SIMD Hardware for Experimentation

The choice of hardware platform for experimentation is important for providing results that align with the goals of the research. While a system could be designed for research using an FPGA platform for SIMD, it would add considerable work and time to the project due to design and implementation time of a large SIMD array. It is also more appropriate to use a realized hardware architecture for verification of its functionality. An integrated circuit processor provided by Atsana Semiconductor is the hardware platform of choice for this research, due both to its ease of use and low power design. Atsana's J2210 architecture falls somewhere in between the smaller scale SIMD CPU instruction set extensions

and large scale processor arrays. The architecture relies on an ARM922T™ CPU as the host and main workhorse of the system. In addition to the ARM, the J2210 contains a two dimensional SIMD array processor similar to the previously described SIMD architectures. It features 96 8-bit Computational Units (CUs) made up of bit serial PEs with control circuitry. The CUs are arranged in a 24×4 two dimensional grid. Direct data communication is performed through memory reads of neighbouring CUs.

The small package, low cost, and low power consumption of the J2210 make it useful for applications in mobile communications and other portable digital devices. The array processor architecture has been used for still and full motion digital image capture, processing, and compression. The goal of this work is to extend this flexible array processor and system to real time 3D graphics rendering. This research will explore the suitability of the Atsana J2210 and SIMD architectures for 3D rendering in embedded low power applications.

2.1.2 Prior Use of SIMD and Graphics

Prior academic work in this area is largely based on the Pixel Planes architecture created by Henry Fuchs [4]. Fuchs' work revolved largely around using a SIMD style architecture where each processor controls an allocated portion of the frame buffer. The pixel planes architecture was designed for high end rendering using expensive hardware, before the availability of consumer end 3D rendering hardware. Fuchs designed his architecture around a 1280x1024 pixel frame

buffer, making a configurable amount of processing elements varying from sixteen thousand to three hundred thousand available. This number of processors meant each would control no more than one hundred pixels, and as few as a four. The 96 computational units in the J2210 may render his algorithms unsuitable for this application. However a similar divided frame buffer method of rendering has been tested on the J2210 processor in chapter 4. The embedded target means lower resolutions and fewer polygons, meaning the algorithm could be appropriate.

Similar work was done by Michael F. Deering of Sun Microsystems while designing accelerated frame buffer memory [5]. Deering's work also used an intelligent frame buffer algorithm with simple processors embedded in the memory array. While not programmable or as flexible as pixel planes, it provided acceleration through an automatic depth buffer test at each pixel write, and single cycle alpha blending. A similar approach could be taken using the J2210 array processor as an intelligent frame buffer. Writes can be buffered and an array processor program can handle depth comparisons and alpha blending. This approach would not harness the full computational power of the CUs but will require the least amount of modification to a sequential algorithm.

2.1.3 SIMD Low Power Applications

The difference between our work and prior research is largely the target application. Previous SIMD graphics renderers have been targeted at high power

graphical systems, but have been made obsolete by higher clock frequencies and more hardware with programmable stream graphics pipelines. High frequencies and high transistor counts are not desirable in a low power, low cost device. By using a programmable SIMD approach we can solve the problem intelligently and with low silicon area cost and low energy consumption. The programmability of the SIMD array makes it more flexible than fixed function algorithms of traditional rendering hardware.

What makes this research necessary and interesting is an increased demand in mobile applications, such as cellular phones. Mobile applications stress the importance of low power consumption due to limited battery life of portable devices. However, performance is rarely an acceptable trade-off to power consumption, as many consumers will not accept lower quality at the gain of longer battery life. The goal is to test the system for suitability of maintaining state of the art effects while consuming lower power than other architectures. A modern stream processor for graphics rendering will surely outperform a the simpler SIMD architecture, but the transistor count and clock frequency will hinder the products' battery life. A flexible middle ground is an interesting area of study relevant only to mobile applications at this time. Desktop processors are drastically increasing in power as they boost performance, with current generation architectures consuming 175 Watts and are rapidly approaching 200W and beyond [6]. Higher power consumption allows the stream processors to increase

performance by running at higher clock frequencies and including more parallel processors. This luxury is certainly not available in the mobile architectures.

In addition to energy and speed, physical silicon area is an important issue in embedded applications. In parallel hardware, there is a linear trade-off between the number of processing elements and performance for a well designed algorithm, assuming sufficient parallelism following Amdahl's Law [7]. The number of processing elements is also directly proportional to the physical area on a chip that the array processor will consume. Chip area affects production costs greatly, since the number of chips per silicon wafer and the chip yield are directly, but not necessarily proportionately, affected.

2.2 Processor in Memory Architectures

Processor in Memory (PiM) architectures are built on the concept of placing microprocessors inside a memory array. These processors are usually small transistor count designs in order to be placed physically at the base of memory columns. This concept exploits the fact that the memory bandwidth is much higher inside the actual memory array, and has been shown to be up to four orders of magnitude higher [8]. This architecture is well suited to SIMD due to the inherent layout of processors and their own private memory. PiM also provides spatial locality for nearest neighbour transfers between processing elements, allowing fast communication between adjacent processing elements.

These architectures tend to be successful in extremely data intensive applications such as image and video compression and manipulation. The Pixel Planes architecture mentioned above uses a similar PiM concept, but with a DSP coprocessor provided for each 128×128 Pixel Processing array. Another inherent advantage of PiM is energy consumption. RAM is accessed by charging all the bitlines in the memory array, and uses less than 1% of those bits, based on the ratio of active bit lines to data pins. If a PiM SIMD architecture is using the entire row, there is no lost efficiency. PiM also provides energy savings due to shorter data buses between the memory and processor.

The J2210 Array Processor uses a smaller scale PiM architecture most similar to C•RAM, developed at the University of Toronto [9]. C•RAM places simple bit-serial processors at the base of every memory column. It uses a few single bit registers in order to perform ALU operations on the data inside its column, which becomes its private memory. C•RAM also uses a linear communication network between neighbouring PEs. The J2210 AP takes this implementation and extends it to use 8 bit wide columns with simple 8 bit registers and ALU.

2.3 Development Platform

The development platform for this project is the J2210 Customer Evaluation Board (CEB) provided by Atsana Semiconductor Corp. It features the J2210

media processor, as well as several other complimentary peripherals. The CEB is shown in Figure 2.2.

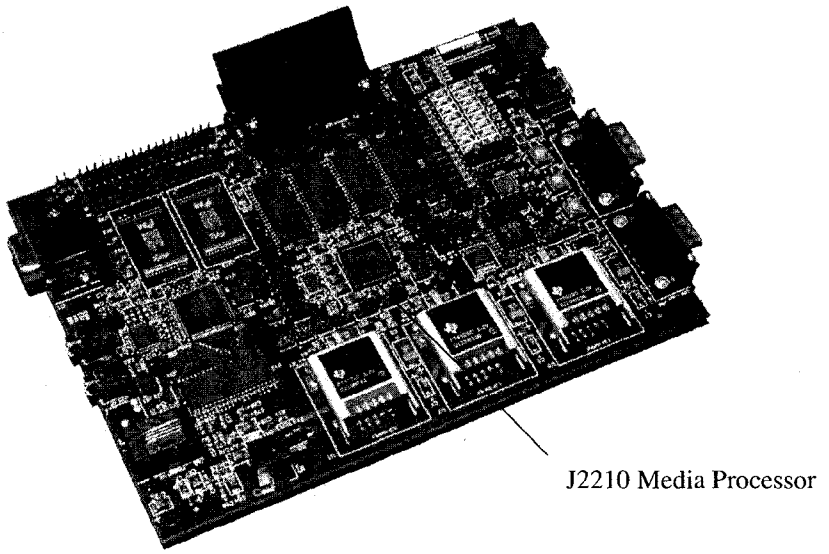

Figure 2.2: Atsana J2210 Customer Evaluation Board

The J2210 contains an ARM922T™ ™ microprocessor which serves as both the sequential processor and host of the SIMD array. The ARM has full access to the memory in the SIMD array. The role of the ARM is to move data in and out of the SIMD array, perform computations best done by a sequential processor, and invoke the array controller programs. The ARM is also free to do its own computations concurrently, which can support the SIMD software, or run a different task.

The ARM sequential processor provides a reasonable benchmark comparison device since this is precisely the type of processor found in embedded

devices. The ARM Development Suite provides C++ compilers and linkers for executables to run on the target hardware. The ARM code for this project is written entirely in C and C++.

The array processor uses a proprietary assembly language called the SIMD Execution Language (SEL). The SEL language uses direct access to the registers and memory variables, and is compiled by a proprietary tool provided by Atsana.

It is important to note that the array processor consumes less energy per operation than the ARM host, which makes the parallel algorithm beneficial for low power operation. As noted above, the power advantage is a result of charging and using all bitlines in the memory array simultaneously, instead of charging a row and only using one word as a sequential processor would.

The J2210 Array Processor contains 96 computational units suitable for integer operations. Typical integer operations are available, addition, subtraction, multiplication, shifts, and logical operations. Floating point operations are not available.

The SIMD array uses uniform memory addressing and uniform network communication. The CU instructions are sent by the array controller which holds and fetches a set of programs for the array. These programs are written in the Atsana created SEL language.

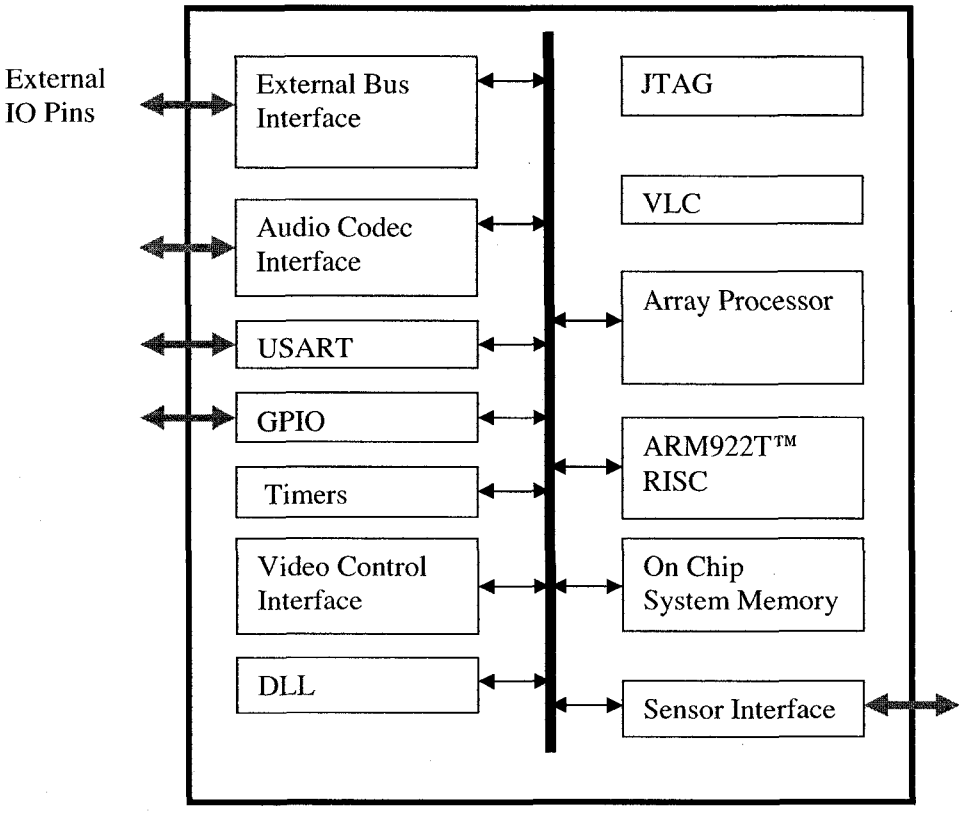

Figure 2.3: J2210 High Level Block Diagram

The J2210 tool kit features a descriptive profiler and simulator which can be used to obtain results for the SIMD algorithms. It provides cycle counts and power consumptions per algorithm and per instruction executed. This allows different SIMD algorithms to be compared in terms of execution time and energy consumption. Execution performance for the algorithms are compared against the ARM processor while running on hardware using an on-chip millisecond timer. Power consumption is measured by the current input to the J2210. By putting the

ARM to sleep while the array processor is executing it is possible to compare the power consumption of each of the processors executing.

2.4 3D Rendering Process Overview

The rendered scenes are created from arrays of vertices containing the information required for the processor to convert them into a 2D representation of the 3D scene. 3D rendering requires that the graphics hardware be powerful enough to create a scene built up from geometric points in real time. In PCs, before Graphics Processing Unit (GPU) hardware was common, 3D rendering was performed with the CPU, but even a powerful sequential processor will struggle with operations required for pixel processing due to the high memory bandwidth required for filling thousands of patches per second in complex scenes.

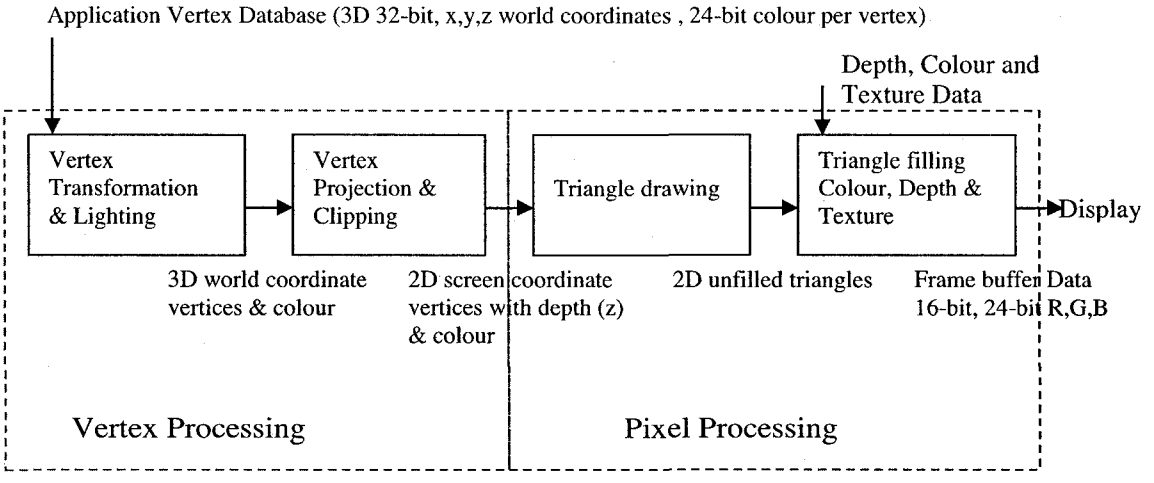

Figure 2.4: 3D Rendering Process Overview

The 3D rendering process is roughly split into two distinct parts, vertex level processing and pixel level processing, as shown in Figure 2.4. Traditional sequential processors are generally suitable for vertex level processing. The amount of memory accesses in the vertex stage is significantly lower than in the pixel stage, so memory bandwidth is less of an issue than with pixel processing. The operations performed on the vertices are largely floating point matrix operations [10], so the floating point coprocessor found in most CPUs provides significant performance increases. Since vertices can be processed largely independently of each other, SIMD instruction extensions in modern processors, such as Intel's SSE1/2, can drastically speed up the calculations. A CPU that can calculate four floating point operations simultaneously can process vertices at nearly four times the speed. However, this powerful processing engine will quickly hit its bottleneck–the pixel calculations.

The process includes a transformation module for geometric translation, rotation, projection and scaling operations. The transformation engine operates on arrays of vertices which are the input to the system. They are points which describe some three dimensional object in local space. At this point the object is described by vectors in space with no image information. Once the vertices have been transformed they are mapped to coordinates for the rasterization phase. The rasterization procedure is the act of converting the vector input data into a two dimensional bitmap for the display. By using the data provided at the vertices, the

data is interpolated across the surface of the polygon when rendered. Typical vertex data includes depth, colour, and texture coordinates. The transformation engine uses floating or fixed point arithmetic.

2.4.1 Implementation of the 3D Rendering Process

The software transformation engine for this project is not speed critical since its performance is not being measured. It is merely used as an input system to both rasterization engines. The rasterization engine is written both for the sequential host processor and the SIMD array, allowing comparison between the two in terms of performance and power. This is possible due to the limited arithmetic required by the rasterization process. It consists mostly of memory operations and some interpolation through multiplication and accumulation, easily performed by both the sequential processor and the SIMD array. In order to achieve useful results the sequential version is designed to achieve peak performance on the ARM processor. Since the ARM and the SIMD array have different execution methods the algorithms used will be different.

Sequential processors can efficiently fill a triangle using a triangle scan conversion algorithm [11]. The edge equations of the triangle are tested at each pixel to determine inclusion. The algorithm simply steps from one pixel the triangle to the next filling pixels until it detects an edge, then moves to the next scan line. The edges can be pre-computed to increase the efficiency, using the

Bresenheim algorithm [12]. This solution can be transferred to a parallel algorithm in a variety of methods, and each must be considered.

Since the work is shared between two processors, the relative impact of the communication between them is an important consideration. For example, the host processor is required to supply data to the SIMD array. If the organization and movement of this data is too large, the time taken approaches the time taken for the host to do the calculations itself. This means that the data preparation has to be minimal, and the algorithm has to be efficient in terms of communication.

2.5 Low Power Embedded Rendering Applications

The target implementation for this SIMD architecture is a mobile handset phone, personal digital assistant, or similar battery powered portable device. The J2210 itself was designed for accelerating real time image and video encoding and decoding at greatly reduced power. While the processor is designed for multimedia applications, we show that the SIMD array processor itself is suitable for pixel operations in a 3D rendering system. It is important to consider the application target when considering performance and power consumption.

A typical media processor today features an embedded RISC host processor accompanied by a number of peripheral interfaces, embedded memory and finally some form of hardware accelerators.

The Atsana J2210 is a system on a chip design, which uses a general purpose ARM922T™™ RISC embedded microprocessor and a fully programmable array processor (AP) for low power multimedia processing. The array processor provides a low energy per operation architecture that provides more computation for a given capacity battery than other embedded systems.

It is important to consider the target application, and how this work will differ from general 3D rendering research. The mobile application target will change various elements of the data being processed. The largest difference between a mobile solution and a desktop solution is screen size and resolution. Physical dimensions of the screen are significantly less than the minimum 15" of a desktop or television display. Reduction in screen size reduces the requirements for geometric detail in a scene, lowering the overall polygon count. Texture detail is lower, resulting in a lower bandwidth requirement for texture transactions. Colour depth is often lower; PC displays are generally 8 bits per colour, while a portable device will likely have 5 bits precision per colour. The most significant difference is screen resolution which has a significant impact on polygon fill rate. Three geometric points on a high resolution display can describe a triangle with hundreds of pixels, while the same points on a low resolution display will only have dozens. The time to fill a polygon increases at roughly a square of the dimensions.

Embedded displays tend to use defined fractions of larger display standards. The Common Intermediate Format (CIF) resolution describes a 352x288 resolution, and a common embedded display resolution is Quarter Common Intermediate Format (QCIF) representing ¼ of the CIF resolution. Similarly higher resolution embedded displays tend to use the Quarter Video Graphics Array (QVGA) resolution of 320x240.

The small package, low cost, and low power consumption of the J2210 make it useful for applications in mobile communications and other portable digital devices. The array processor architecture has been used for still and full motion digital image capture, processing, and compression. The goal of this work is to extend this flexible array processor and system to real time 3D graphics rendering. By benchmarking these algorithms the bottlenecks of the system will be identified. These results should provide enough information on how modifications to the system could improve performance. These modifications are identified as the issues arise, and their impact on the system is analyzed.

2.6 Summary

This project uses a hardware platform that combines concepts of PiM and SIMD architectures with the goal of low power consumption. The target application is computationally expensive real time 3D rendering. The goal is to produce a low power, efficient rendering algorithm that is suitable to the J2210 array processor.

This chapter provides the background information necessary to step forward and combine these concepts into a SIMD rendering solution.

Chapter 3
Data Parallelism in 3D Rendering

In order to determine how to accelerate the 3D rendering process on the J2210 architecture, the process must be broken down into stages. The overall view of the graphics rendering is broken into two major portions, vertex level processing and pixel level processing. Vertices describe the objects in three dimensional space, and conceptually represent a wireframe description of the scene. The task of taking these points and making them solid and textured is the pixel level processing. Both major portions offer parallelism which can be taken advantage of with SIMD hardware.

We will show that the vertex stages are not well suited to being accelerated on the J2210, but the pixel stages show promise for a SIMD implementation on the J2210 Array Processor. A process diagram with a rough breakdown of pixel and vertex stages is shown in Figure 3.1.

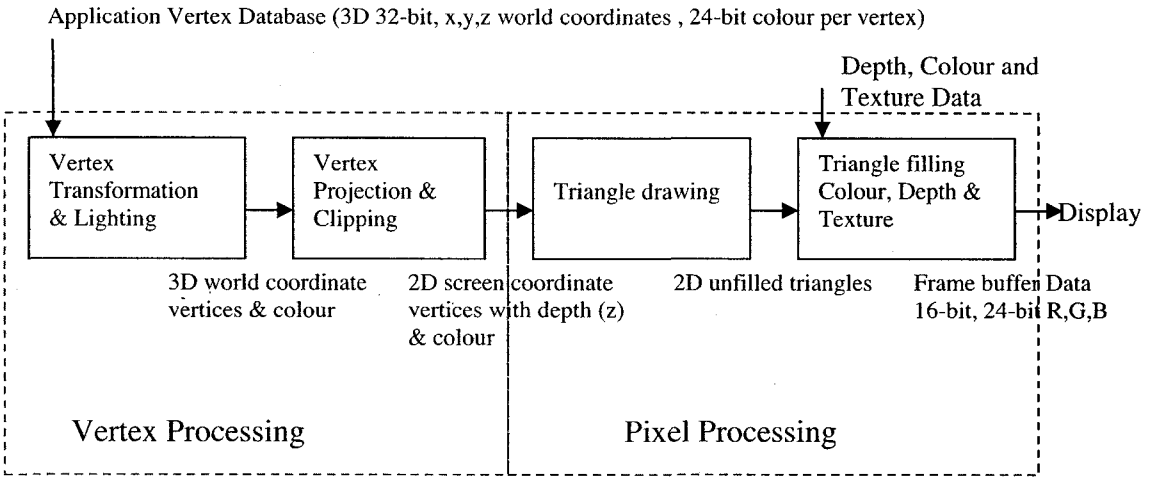

Figure 3.1: 3D Rendering Process Breakdown

3.1 Vertex Processing Data Parallelism

In a rendered scene, all objects begin as a data set of vertices. Often applications load this binary data from a file and store it in system memory as it is needed. This vertex data is an array of points in a three dimensional space, and it is generally static floating point data. Since the data is static, there exists no dependency on the previous state of the data, each time the scene is drawn, the data is reloaded from its original state and the corresponding transformations are applied without any prior knowledge. In addition no object has any data dependency to any other object in the scene, since each element will be transformed independently. Objects can represent anything, from a box with a few vertices, to an entire landscape with millions.

3.1.1 Vertex Transformation Stage

The transformation that these data sets go through to become part of the scene is a series of vector by matrix multiplications. Each point becomes a vector (a three dimensional point in space). The physical orientation of each point is changed through a transformation matrix. The transformation matrix will describe the operations that will be applied to each point in the objects. The transformation matrix is formed by combining a series of matrix operations, which are generally translation, rotation and scaling. Since one matrix can describe a combination of operations the transformation matrix for an object is only calculated once. Once the final matrix calculations are finished, they will be applied to each vertex in the object. Since the objects are generally rigid, the same matrix will be applied to all points in each object, independent of each other. A matrix by vector operation becomes a series of multiplications and additions. Applying a matrix to a single vertex is a simple operation.

$$\begin{aligned} x_{trans} &= x \times M_{11} + y \times M_{21} + z \times M_{31} \\ y_{trans} &= x \times M_{12} + y \times M_{22} + z \times M_{32} \\ z_{trans} &= x \times M_{13} + y \times M_{23} + z \times M_{32} \end{aligned} \tag{3.1}$$

It can be seen that even for one point, there is 9 individual multiplications which have no dependencies, followed by 6 additions, which depend only on the multiplications before them. In a theoretical system with 9 parallel multipliers and 6 parallel adders, this is a two stage process to perform the complete matrix

by vector multiplication. The parallelism goes much beyond this simple single vertex scenario. The actual loop, shown below, has many iterations of the same operation.

for each of n vertices
{

$$\begin{aligned} x_{n\,trans} &= x_n \times M_{11} + y_n \times M_{21} + z_n \times M_{31} \\ y_{n\,trans} &= x_n \times M_{12} + y_n \times M_{22} + z_n \times M_{32} \\ z_{n\,trans} &= x_n \times M_{13} + y_n \times M_{23} + z_n \times M_{32} \end{aligned} \tag{3.2}$$

}

In this loop it is apparent that there is no dependency between the current and previously calculated vertices. At this point, if there was a processor with 9 by n multipliers and 6 by n adders it is still a two stage process. The parallelism is only broken when the matrix must be changed. In practice, most objects are static relative to the viewer, and they will all be applied by the same matrix. Only the other objects which have their own movements need a different transformation matrix. Generally since most the objects have hundreds of vertices as a minimum the actual matrix generation is a very small part of rendering, so even with hundreds of parallel multipliers, the usage of the multipliers and adders would be near 100%.

3.1.2 Perspective Transformation Stage

Moving beyond the transformation matrix, the perspective matrix is applied. The perspective matrix is what makes the scene mimic the human eye; as objects become further away, they become smaller. The perspective matrix is the same

for all objects in the scene, regardless of transformation, which further pushes the capability of parallel vertex calculations. The following is an example of a simple perspective projection calculation on the x and y coordinates

$$q_x = -d\frac{p_x}{p_z}$$
$$q_y = -d\frac{p_y}{p_z} \tag{3.3}$$

where q is the output vector, p is the input vector and d is the distance of the projection plane from the camera [13].

3.1.2 Lighting Stage

Lighting operations are similar. Each light in the scene is applied to each vertex via a distance calculation. Several independent multiplications are added together to form a lighting effect, then divided by a distance calculation which depends on the properties of the light. As with transformations, all operations are vertex independent. Again, a simple lighting equation for point lights demonstrates that a division by the distance from the light is required. Here,

$$C = \frac{1}{k_c + k_l d + k_q d^2} C_0, \tag{3.4}$$

where C is the colour of the light, and k_c, k_l and k_q are the constant, linear, and quadratic attenuation constants of the lights respectively [14].

3.1.3 Vertex Acceleration on the J2210

In most current rendering systems, the transformation, perspective and lighting stages of the process are calculated using floating point precision. Repeated multiplications and divisions quickly break down the accuracy of a fixed point representation. Unfortunately this dependence on the precision of floating point data makes the J2210's Array Processor inappropriate for vertex level parallel processing. While it is possible to implement vertex processing in a fixed point implementation, the 8-bit CU architecture on the Array Processor does not provide enough precision with its instruction set, which has at most 24 bits of precision with costly repeated multiplications. In addition the AP does not have the instructions needed for division required by the perspective correction stage.

Apart from the mathematical issues, the data formatting presents another problem with vertex acceleration. The end result of the first stages does not leave data ideal for storage in the array processor. Several vertices in screen coordinates are scattered in the memory of the array processor. This data has to be sorted to be useful for triangle drawing, which requires unloading it all to the ARM for sorting, then loading it back, or having the array processor to do sorting, which would require many steps with nearest neighbour data transfer, hindering the effectiveness of this approach. Considering that each CU could contain data to be placed anywhere in the array, there could be up to 96 transfers in each direction, or 192 copies per byte, in addition to a small program to run with each

copy step, to determine if each CU is the destination. In addition to the raw data being copied, some destination information is required to be transferred as well. The pixel stage of rasterization is much more suitable to the architecture.

3.2 Pixel Processing Data Parallelism

In early real time rendering when the scenes were traditionally simple in the geometry states the pixel level calculations were the bottleneck. At this point pixel operations were all integer data, but even SIMD CPU extensions could not provide enough power to fill triangles at high resolutions with acceptable performance. Pixel operations are relatively independent of the input data. A triangle consisting of three floating point vertices can actually represent a few pixels, up to millions of pixels. The number of pixels in a triangle goes up quadratically as the linear resolution of the display increases. This is why early graphical applications which ran without a hardware accelerator were generally run at low resolutions like 320x200. A triangle which represents 200 pixels at 320x240 could represent 800 pixels at 640x480 and 1600 pixels at 1280x960. Increasing the resolution by a factor of two in each dimension increases the processing time by a factor of four.

3.2.1 Attribute Interpolation

In order to fill the 2D frame buffer with the proper output colour data, several components need to be calculated on a pixel by pixel basis. Since all of the

information in a polygon is stored in the vertices which define its outer points, all of the data inside must be interpolated between those points. Colour data, represented individually in red (r), green (g) and blue (b) values is the lighting information for each pixel, based on the coloured value of both the polygon itself, as well as the intensity of the lights which are hitting it. Depth values (z) are required in order to evaluate which objects will be on top of each other in the two dimensional scene. When the polygon has a material surface, a texture must be applied. Texture coordinates are interpolated in a similar fashion to find the x and y index (s, t) into the 2D texture that is applied to the polygon, as shown in the pseudo code below.

```
for each pixel n
    // Interpolate the values using the invariants
   interpolate depth z
   interpolate lighting r, g, b
   interpolate texture coordinate s, t

   // find texture colours using the interpolated texture coordinates s,t
   fetch texture r, g, b

   // modulate the texture r,g,b with the previously interpolated r,g,b
   combine lighting and texture r, g, b

   // if the pixel is closer than the previously written pixel, write it
   if z < zcurrent
      write back r, g, b, z
```

Table 3.1: Pseudo code for pixel processing

The interpolation operations are a method of taking the lighting colour values at each of the three vertices in a triangle and applying them to a pixel somewhere within that triangle. It is a combination of integer multiplication and addition. Combining the lighting and texture colours is a weighted multiplication.

The integer precision of the pixel processing makes it much simpler than floating point vertex operations. The sheer number of integer operations hurt the performance, but a more significant cause for poor performance of pixel processing by a standard CPU was the high memory bandwidth required to keep up with the operations, which is well described in the Graphics Hardware chapter of Real Time Rendering [13]. For each pixel in the frame several values must be read from and written to system memory, and the buffers are so large that the cache becomes less useful, due to large variances in spatial and temporal locality of the pixels being operated on. The graphics cards addressed this by putting private texture and frame buffer memories on the board with high bandwidth connections to the graphics processor. This high processor to frame buffer bandwidth requirement can also be addressed using a SIMD Processor in Memory architecture such as the J2210 AP.

There are two specific aspects of the Array Processor which can be used to accelerate the pixel portion of the 3D rendering, memory bandwidth and parallel arithmetic. Power and speed measurements on both algorithms are calculated using the method described in chapter 4.

3.2.2 Tile Based Parallelism

The first method is described as a tile based method. This method is most similar to the Pixel Planes architecture described in chapter 2, in which each Computational Unit is responsible for a portion of the frame buffer, and when

combined they form the full image. While the Pixel Planes architecture had thousands of processing elements, the Array Processor has only 96. Where Pixel Planes was designed for speed and high precision scenes, the J2210 is designed for mobile, low power applications. Instead of a per-pixel, Phong shading approach [15], this implementation uses the more traditional linear interpolative Gouraud method [16] described above. The basic idea is to divide the frame buffer into 96 non-overlapping rectangles, each of which is assigned to a CU. The CU is responsible for filling in its own piece, while the other CUs do theirs in parallel.

The array processor uses uniform memory addressing, which means that each CU in the array must read from or write to the same address in memory for any operation. As well there is no opportunity for conditional execution since any element in the array may be processing useful data, so each CU must always process each pixel inside of its portion in the frame, regardless of whether or not that pixel is inside the triangle it is currently working on. The layout is shown visually in Figure 3.2.

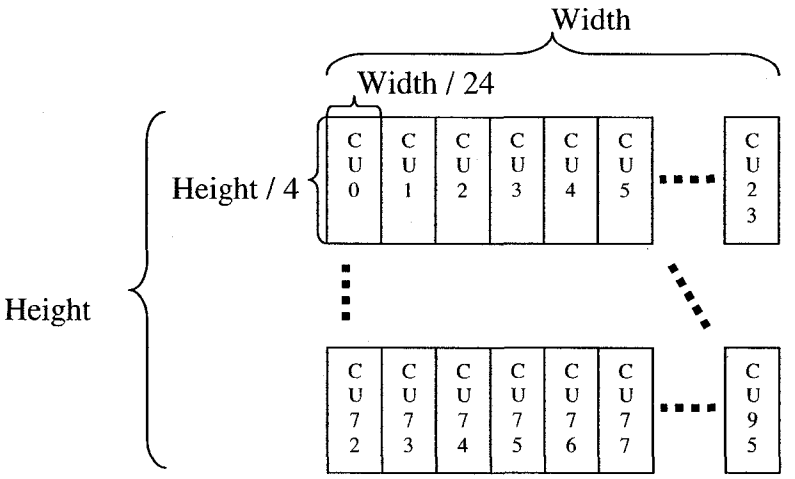

Figure 3.2: Pixel layout in the array as it forms the complete frame buffer.

Since the array is already laid out in a two dimensional fashion, the frame buffer will be divided evenly in order to match. The basic idea is for the algorithm to run once for each triangle that can be calculated without dependency on another CU. A peak calculation rate of 96 independent triangles can be shaded per pass, if they are each contained within unique CUs. Of course if there is one very large triangle, which takes up a significant portion of the array, the rate drops off significantly, as low as one triangle per pass. However, as triangles shrink in size, the parallelism increases. If a triangle crosses a single rectangle boundary it immediately doubles the amount of processor time dedicated to that triangle, since it must be processed by two CUs. Therefore the smaller the triangles are, the less likely they cross boundaries. The implementation and results of this algorithm are described in detail in chapter 4.

3.2.3 Pixel Based Parallelism

The second approach takes a more computational approach to the problem. Since each pixel inside of a triangle can be calculated independently of all others, they can all be calculated in parallel using only the interpolation values calculated from the vertices which describe it. This approaches uses the J2210 Array Processor as a 96 input ALU, calculating 96 pixels with each pass. Each CU runs a simple program to calculate the attributes required to write that pixel to the frame buffer, as shown below. The inclusion test is done before hand by the ARM processor.

```
for each pixel n
   interpolate depth z
   interpolate lighting r, g, b
   interpolate texture coordinate s, t
```

Table 3.2: Pseudo code for CU interpolation

While not strictly a requirement, it is easiest when it works on a single triangle with each pass. Therefore a triangle with 96 pixels or less would take one pass of the algorithm, while larger triangles would require more, based on multiples of 96. Each interpolation step is basically two multiplications and two additions. The calculation of the red value of any pixel is as follows,

$$R_{xy} = R_0 + x\frac{dR}{dx} + y\frac{dR}{dy} \tag{3.5}$$

where dR/dx and dR/dy are constant for the triangle. This calculation of R can be substituted for all of the other attributes. In order to estimate the feasibility of this algorithm on the J2210 Array Processor, we must compare the measured performance of addition and multiplication of the two architectures. The

following table depicts the measured comparison of addition and multiplication, using two values from memory, and writing the result out to memory. The following results are measured in real time from the hardware in continuous loop on the operations. The power measurements are done using the arrangement described in the power section of chapter 4. The Energy measurement includes both the Array Processor consumption as well as the Array Controller and CMEM Interface Unit required for it to run. The method for energy consumption measurement is described in chapter 4.

	Millions of 16 Bit Additions per Second	***Energy Per Addition (nJ)***
ARM 922T (192 MHz)	13.9	4.30
Array Processor (96MHz)	1498.5	0.039
Ratio (AP to ARM)	108×	0.01×

Table 3.3: Breakdown of addition operations on ARM and AP

It can be seen that the 96 CU array processor exceeds the ARM922T™ by more than a factor of 100, both in terms of performance and energy consumption. Due to the parallel nature of the Array Processor, the speed and energy consumption of a single addition would be very similar to the consumption of the ARM. The increased performance requires a full array of data to work on, which is why this platform should be suitable to the parallel nature of pixel rasterization. Multiplication is implemented in the CU, by Atsana, using repeated binary

additions. Since it requires several additions to calculate the result, the performance increase is not as great, but still apparent.

	Millions of 16 Bit by 8 Bit Multiplications per Second	***Energy Per Multiplication (nJ)***
ARM 922T (192 MHz)	13.9	4.37
Array Processor (96MHz)	87.7	2.92
Ratio (AP to ARM)	6×	0.67×

Table 3.4: Breakdown of multiplication operations on ARM and AP

Due to the fact that the implementation of multiplication by the Computational Units is serial addition, the result is roughly 17× slower than the addition, however the fact that there are 96 in parallel, still allows the array processor to be faster than the ARM by a factor of 6. As well the energy consumption remains much lower.

The fact that power and speed advantages are available for both the addition and multiplication shows that it is feasible to treat the array processor as an ALU for the attribute interpolation portion of the 3D rendering process. The implementation and results of this approach are described in chapter 5.

3.3 Summary

The 3D rendering process has significant data parallelism which can be readily exploited with a SIMD architecture since the data inputs are largely independent of the outputs. While vertex processing was found to be relatively inappropriate for the low precision integer architecture of the Atsana J2210 array processor, pixel processing has several appropriate stages.

The first approach is a tile based frame buffer system in which the Array Processor acts as a series of processors inside the frame buffer, drawing the data directly into the memory.

The second approach uses the Array Processor as an embedded massively parallel ALU for addition and multiplication operations present in the attribute interpolation stage.

Chapter 4

Tile Based Rendering Approach

The 3D rendering process presented previously contains several opportunities for parallel processing. The SIMD array in the Atsana J2210 provides 96, CUs which would best be fully utilized in order to maximize the performance gain of the parallel algorithm. In this algorithm the array processor performs the latter half of the rendering operations, beginning with the triangle setup. We will show that with the right data the tile based algorithm can show up to a speed increase of 10.5× over the ARM with 8.4× less energy, but that performance increase can break down rapidly depending on the data input. The division of work is shown in Figure 4.1.

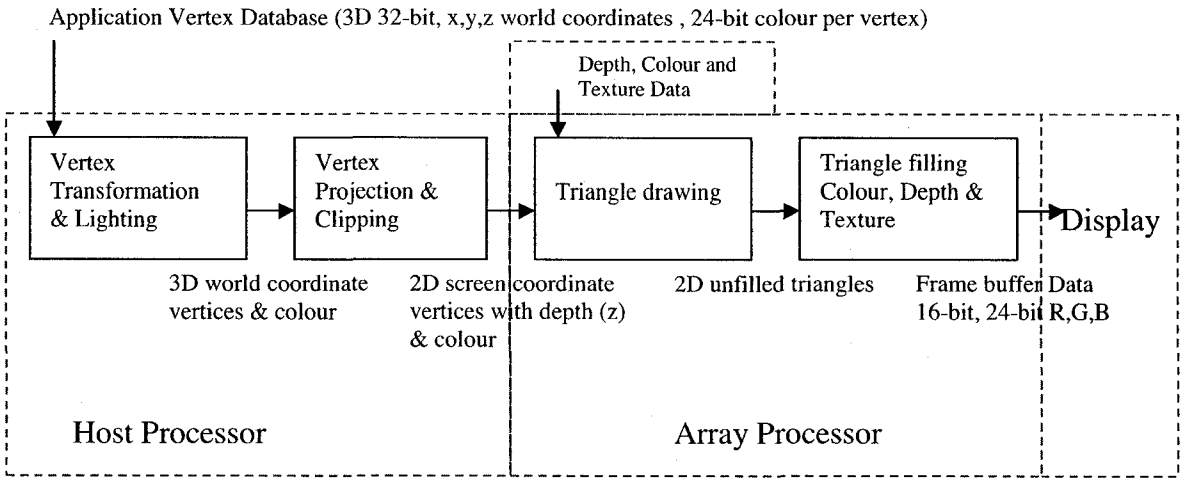

Figure 4.1: Division of Rendering work

Since, in this implementation, the SIMD memory contains the frame buffer, pixel level data is useful as the end result, since it can be copied out directly to the external frame buffer for the display. The frame buffer will be divided evenly among the 96 CUs, so that each CU carries a specific number of pixels of the output image as shown in Figure 4.2. It will hold the RGB colour data and depth value. The frame buffer size in this implementation is 192×144, which is small, but reasonable for a hand held system that the J2210 is catered towards. This is slightly larger than the typical mobile QCIF display size of 176×144.

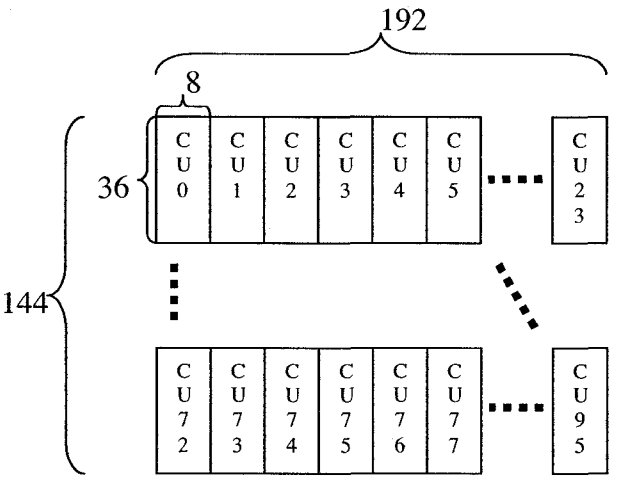

Figure 4.2: Pixel layout in the array as it forms the complete frame buffer.

With each CU taking a portion of the frame buffer, filling algorithms will be equally divided in screen space. If a triangle is contained within 4 CUs, it leaves 92 other CUs free to work on other triangles. As triangle sizes decrease, the maximum amount of parallelism is created.

The line equations that describe the triangles are copied to the CU from the ARM host processor. The filling is done such that each CU tests if each of its pixels are within the triangle boundaries, and set their write enable bit, for masking. After a pixel is determined to be inside the triangle additional masking step is required. The depth test will resolve the relative depth of the pixels, which is used to determine the pixel's visibility, and thus draw mask. Once the pixels are determined to be included, colour interpolation is done on the processor array using an implementation of the Gouraud shading algorithm described in Chapter 3

to combine colours smoothly across the face of the triangle. After shading, if texturing is enabled, the texture coordinate for each pixel is determined. Texture lookups are not directly possible using a SIMD architecture with uniform memory addressing, but there are workarounds described in the texturing section in this chapter. This texture pixel, or texel, is applied to the existing colour value of the pixel, usually with a normalized product between the texel colour value and the shading colour value. For each colour, the value of the pixel and the texel are multiplied, the product is then normalized, as if both were scaled between 0 and 1. However, in the integer case they are scaled between 0 and 255, and the result is divided by 256 via an 8 bit right shift. On a perfectly white lit triangle, the full texture will appear, but if the lighting is red only, only the red components of the texture will show through.

Since the CUs have no support for floating point arithmetic, the goal is to have an integer based algorithm for testing triangle boundaries. The traditional **y=mx+b** line equation will likely require a floating point slope and intercept, since there is no coefficient on the **y** variable and **m** can go to infinity. This equation can be converted to **Ax+By+C=0** which can express the same information using only integer coefficients. The conversion is done by the ARM before supplying the **Ax+By+C=0** equation to the CUs to facilitate the floating point arithmetic required.

The ARM processor can handle the input vertices, and form 3 line equations of the form listed above that will intersect to form a single triangle. Since the ARM can check to see which CUs the triangle will overlap, it can choose to send relevant triangle equations only to the CUs that require them. It can then send another set of equations to different CUs allowing them to process triangles which are relevant to their memory space.

4.1 Host/Array Communication Implementation

The Atsana J2210 Array Processor architecture has an unusual memory layout. Each of the 96 CUs has a private 4KB of memory. The memory is arranged physically as an array of 4096 bytes directly above the processing hardware. Since each CU has a one byte word length, writing to the memory from the ARM has proven to require some care. The function provided in the API is `WriteMem32`, which writes 4 bytes to the address provided. While the simulation API allows `WriteMem8u`, which only writes one byte, this project solely uses `WriteMem32` in the interest of a code base that runs on both the target J2210 and the simulator.

Since `WriteMem32` writes 4 bytes, and due to the physical layout of the memory, the provided data actually writes over the address space of 4 CUs. Writing `0xffffffff` at CU 0,0 will write `0xff` into CUs 0-3, as shown in Figure 4.3.

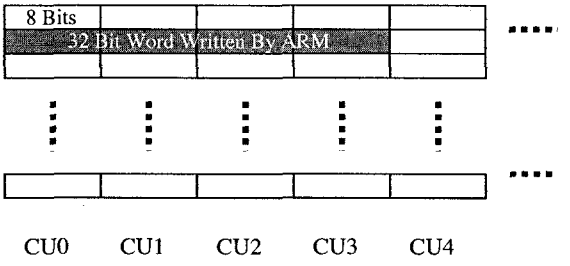

Figure 4.3: Memory layout in the CUs of the memory array.

In order to write different values into the address space of neighbouring CUs, many bitwise operations must be performed. In order to write to the correct CUs, the data can be packed in advance using the code shown below. In order to have accurate values for the line equations, signed short integers (16 bit) are required, which means that the variable must be further broken down and placed in two successive locations. An example of writing 4 signed short integers into 4 successive CUs is shown below. The first group is the most significant 8 bits of the variable, followed by the least significant bits. The code is iterated such that '`j`' iterates through all rows, and '`i`' iterates through every fourth column. The first packed value written in the code below is the most significant byte of the short integers, and the second value is the least significant byte.

```
for(i=1; i<= ArrayWidth; i++)
{
  for(j=1; j<= ArrayHeight; j++)
  {
       temp = ((start1>>8) & 0x000000ff) |
              ((start2) & 0x0000ff00)    |
              ((start3<<8) & 0x00ff0000) |
              ((start4<<16) & 0xff000000);

       WriteMem32(data_addr, i, j, temp);

       temp = ((start1) & 0x000000ff)    |
              ((start2<<8) & 0x0000ff00) |
              ((start3<<16) & 0x00ff0000)|
              ((start4<<24) & 0xff000000);

       WriteMem32(data_addr+1, i, j, temp);
  }
 }
```

Table 4.1: Code for packing and writing short integers to the Array Processor

Retrieving data from the array is much simpler. Although ReadMem32 operates much in the same way as WriteMem32, the ARM host can cast the retrieved integer type as char, truncating all but the relevant byte.

In terms of communication, the host provides all of the data for the SIMD programs through WriteMem32. It provides the line equations for triangle drawing, and vertex colour for the Gouraud shading algorithm. In addition to the data to be processed, the ARM code provides each CU with other useful data listed below.

- Its physical location in the array at `data[XPOS]`, and `data[YPOS]`
- `0x01` at `data[ONE]`
- `0xff` at `data[FF]`
- Integers `0` through `7` starting at `data[XCOUNTER]` for horizontal pixel counters
- Integers `0` through `35` starting at `data[YCOUNTER]` for vertical pixel counters

The CUs physical location is useful for many nearest neighbour operations. The SEL language cannot assign constants to registers, but can load variables, so useful data like 0x01 and 0xff can be retrieved from memory. While SEL can use its loop variables as array indexes, it cannot assign them to registers, which is why the ARM provides the data in memory. In the 3D SEL operations, the CU pixel memory is iterated through with two '`for`' loops, with values `x` and `y`. Since the `x` and `y` values are often needed, they are grabbed from memory with `data[XCOUNTER+x]` and `data[YCOUNTER+y]` to be used in the mathematical equations.

4.2 Triangle Drawing Implementation

The triangle edges are defined with three equations calculated by the ARM using integer division which the SIMD array is not easily capable of. It creates a half plane equation of the form $Ax+By+C \geq 0$ with signed integer coefficients that the SIMD array can readily deal with. Depending on the placement of the three vertices, points in the triangle will be either under two of the three lines, and over one, or over two and under one. A special case for vertical lines assigns a very large slope, which will evaluate to a vertical line within the pixel space of the frame buffer. The type of triangle is determined by the x placement of the lowest pixel, either between the other two, or to the side, as shown in Figure 4.4.

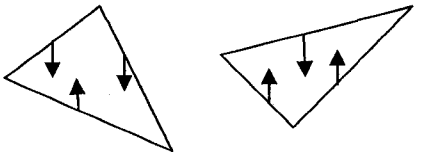

Figure 4.4: Definition of the two orientations of triangles dealt with by the algorithm

The SIMD array calculates the `Ax+By+C` line equation for each point in its local frame buffer using integer math, and checks the sign of the result. The sign of the result determines the pixels location, above, on, or below the line. The coefficient signs can be flipped to reverse the equation. This way, "inside the triangle" can always be set as a positive result to the line equation, simplifying the SIMD code.

Each CU contains a write enable array, which holds the write enable bit for each of the 8×36 pixels that it manages. Initially all of the write enable bits are set to 1. The 1 value specifies inclusion in the triangle. Then, for each of the lines, it calculates whether or not each pixel is above or below the line. For a top line, it will set the write enable of every pixel above to 0. The 0 represents that the pixel is outside of the triangle. For a bottom line, it will set the write enable of every pixel below to 0. The final line will be done the same way, closing the triangle as shown in Figure 4.5. This process masks out the triangle, leaving the write enable bit high for any pixel that is actually contained within. This potentially means a very small portion of the CU is actually producing useful

results, but the entire array of pixels must be processed. On page 49 an optimization for multiple pixels per CU is described.

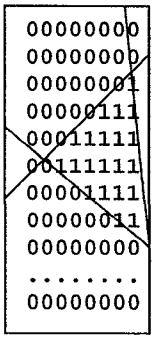

Figure 4.5: Example line intersections forming the write enable array to draw the triangle

Each CU contains a different portion of the image. The 8×36 segments of pixel data have to be set to their absolute location in the frame buffer in order to properly draw triangles that span more than one CU. This is why each CU is loaded with an X and Y offset by the ARM host. The offset gives the absolute location of the bottom right pixel in each 8×36 segment of the screen image. The bottom right is chosen because the SEL language only accepts loops of one format.

```
for(i = CONSTANT; i >= 0; i-=N){ … }
```

The origin of the frame buffer is the top left pixel of CU 0,0. For this design this means the loops have to start in the bottom right pixel and work the way up to the top left.

```
for(y = YPIXELSPERCU-1; y >= 0; y--)
{
    for(x = XPIXELSPERCU-1; x >= 0; x--){ … }
}
```

So each CU begins with its lowest right pixel. The offset has to be added to ensure correct relation to the lines. This requires the use of some of the data provided by the ARM, mentioned in the communication section.

```
// get the counter value (can't set to x)
load counter value
add counter to start offset
```

Then the result is multiplied by the coefficient.

```
// load A (line defines which of the 3)
load A
multiply A by x
```

The 'b*y' term is calculated similarly, and the two are summed, and finally added to the 'c' coefficient. Then the sign is checked to be positive, zero, or negative and the negative values will be masked away.

Performing three multiplications per pixel, per line, results in 864 16-bit by 8-bit multiplications. Each signed 16-bit by 8-bit multiplication takes 40 cycles, making this operation very costly in a real time system. With some analysis, it can be seen that the Ax+By+C line equation need only be calculated only once per CU. Each of the 36x8 pixels in a CU are looped through and calculated. If the x or y coordinates are only changed by a value of one, which

they will within the loop, the A or B coefficient can simply be added to the previous result.

$$A(x) + B(y) + C = A(x-1) + A + B(y) + C \qquad (4.1)$$

For each change in x, the value of A is added, for each change in y, B is added. So, Ax+By+C is calculated once per CU, per line, then for each of the 864 pixels, only one addition operation is performed. The 16-bit addition or subtraction, takes 4 clock cycles, an order of magnitude less than the previous implementation. Now only 2 multiplications are required in the SIMD array per line, or 6 per polygon.

The sequential code contains a function for reading the frame buffer from the SIMD array simulator and dumping the data to an uncompressed Targa (.tga) image. The uncompressed Targa format is a simple image format containing a small header for width, height, and pixel depth information followed by the raw pixel data. Figure 4.6 shows a dumped output of the triangle created by this algorithm set to white. The input vertices were (1,24), (46,2), (57,120).

Figure 4.6: SIMD array painting a white triangle with three intersecting lines.

This triangle spans the first 8 CUs horizontally, and all 4 vertically. Different CUs loaded with different line equations will draw multiple independent triangles simultaneously.

4.3 Gouraud Shading Implementation

While filling triangles with solid colours can create realistic models, a shading technique will result in much smoother transitions. This stage takes the colour value at each vertex, and linearly interpolates along the surface of the triangle to create a smooth colour conversion.

Since any three points in space will lie on a plane, there must be two slopes which define that plane, aligned to the x and y axis. The ARM processor finds these slopes for the SIMD array to shade the triangle. The colour in the current implementation is 24-bits-per-pixel RGB data. The ARM finds the slopes that define the red, green, and blue planes. These values are used in the complete interpolation equation (3.5).

```
area = ((P2.x - P1.x) * (P3.y - P1.y) -
        (P2.y - P1.y) * (P3.x - P1.x));

         // Find the red slopes,
         //8 binary places after point

drdx = (((P2.red - P1.red) * (P3.y - P1.y) -
         (P3.red - P1.red) * (P2.y - P1.y))
         << 8) / area;

drdy = (((P3.red - P1.red) * (P2.x - P1.x) -
         (P2.red - P1.red) * (P3.x - P1.x))
         << 8) / area;
```

Table 4.2: Calculations for fixed point interpolation coefficients

The colour output will always be unsigned integers, but the slopes will likely be small and fractional. The ARM creates these slopes using a fixed point 16 bit short value, with 8 bits for the whole number portion and 8 bits for the fraction. This method creates values that are very easily implemented in the SIMD instruction set. In a 16 bit register, the value of the pixel is stored, and then the delta value is added. If the delta value is less than one, the whole portion may not change, but it may in the next iteration.

```
// Calculate red
load red value
subtract x red delta
store red value

// Calculate green
load green value
subtract x green delta
store green value

// Calculate blue
load blue value
subtract x blue delta
store blue value

// Convert to 16 bit format
combine red, green, blue to 16 bit format

// Store if WE is true
load write enable from array
write pixel colour
```

Table 4.3: Pseudo code for interpolation algorithm

The Gouraud algorithm must also loop through each pixel in the CU and write only to the unmasked pixels. This means that the host must define the

starting value of each CU in the bottom right corner. As each x value increases, the *d/dx* is subtracted, since the loop is traversing from right to left. At the beginning of the next row, the value is reset, and the y delta is subtracted, and x loop repeats.

Figure 4.7 shows the frame buffer contents of the Gouraud algorithm on two triangles that are drawn and shaded simultaneously using different CUs.

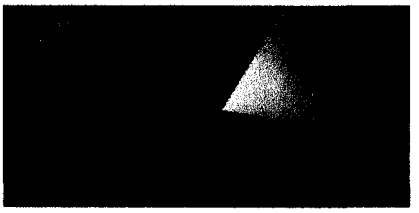

Figure 4.7: Two triangles drawn and Gouraud shaded simultaneously by different CUs.

This program has many work-arounds to the limited address generation unit of the SEL compiler. First, the x loop is unrolled manually. Since the desired output is sequential memory words of RGB data, the array index would need to be fairly complicated. Here BYTESPP defines the 3 bytes for each pixel, and XPIXELS is the width of the image in pixels.

```
pixeldata[x*BYTESPP + y*XPIXELS*BYTESPP + COLOUR] = A2;
```

Since the compiler would not accept this, the code was unrolled to eliminate **x*BYTESPP** and replaced with constants. Unfortunately, while the compiler supports **y*16** in the array index, it does not support the **y*24** required

for **y*XPIXELS*BYTESPP** so initially all work was done in 16 bits-per-pixel. In the current implementation there is 24 bits-per-pixel support, but the blue data is in a separate array. The separate array is useful for verification, but not for video output. To be used in a real time video system, a hardware enhancement will have to be made such that the RGB data is packed together.

4.4 Depth Calculation and Evaluation

One advantage of the Gouraud shading algorithm is that it also works with the depth of each pixel in the triangle. In addition to interpolating R,G, and B values between three vertices, the Z value can also be linearly interpolated. The Z value is actually determined before the Gouraud colour is used at all.

In order to determine whether or not a pixel should be drawn, the depth value is determined using the same equations as the colour calculations above. If the depth value calculated is closer than the previous value, then the new value will be taken. This is performed on the SIMD array by setting a write enable flag if the value is closer. This write enable flag is then combined with the original write enable, from the triangle calculation, with a logical AND. If the pixel is closer, and inside the triangle, its final write enable value passed to the Gouraud algorithm will be set true, as shown in Figure 4.8.

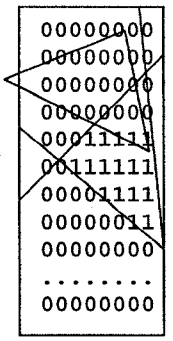

Figure 4.8: The triangle figure above, this time with another triangle masking out a portion

This algorithm allows opaque triangles to be passed in any order to the array processor and guarantees correct output. While in a sequential system the depth test can save a significant amount of calculation, those possibilities are less in this SIMD system. Since the SIMD utilization requires all pixels be calculated unless none at all need to be, the only situation in which the colour calculations could be discarded is if every pixel to be calculated in the array processor was masked out.

```
// Similar to RGB calculations
load z value
subtract x z delta
store z value

// Now determine if closer
if z is smaller, set WE

// Mask with the old write enable value
combine depth WE and triangle WE with logical and
```

Table 4.4: Pseudo code for depth test

The actual number format of this algorithm is irrelevant, since all the numbers are relative to each other. They use 16 bit precision, which represents a 2^{16} level depth buffer. All numbers are considered positive, with numbers closest to 0 being closest to the camera. There are limitations to an integer depth value, as the Z data is prone to precision errors on two surfaces which may be close together, causing an artefact known as "z-fighting". Z-fighting appears as tears in a texture when precision problems cause two different polygons with similar z values to alternate which pixels are drawn. Integer Z will produce many more artefacts than a properly scaled floating point Z.

Figure 4.9: Four Cubes drawn using the depth test and Gouraud shading algorithm

4.5 Texture Coordinate Calculation

Once the colour values are determined, the texture coordinates are calculated using the same method as the colour and depth values. These values are calculated as an index into the texture, so in a 64 pixel by 64 pixel texture, they will be between 0 and 63. They are calculated in an 8.8 fixed point integer format the same way as the colour values in the Gouraud step. The 8.8 fixed value is

stored in a 16 bit short integer, with the fixed binary point position dividing 8 bits of integer and 8 bits of fraction. This allows a maximum texture size of 256×256, which is significantly larger than required, since it is much higher than the display resolution. Typical texture sizes for this display resolution would be 16×16, 32×32 and in highest detail cases, 64×64.

```
// Similar to RGB, only now S & T texture indices
load s value
subtract x s delta
store s value

load t value
subtract x t delta
store t value


load write enable
store s
store t
```

Table 4.5: Pseudo code for texture calculation

While the SIMD array is capable of calculating the index into the texture for each pixel in the triangle, it is not capable of fetching the actual colour data at that pixel, due to the fact that it cannot access external memory. Since the textures are too large to be stored inside the SIMD array, they must be held in the host memory. There are two alternatives to applying the texture colour to the Gouraud colour via the host processor. The first alternative is to let the SIMD array do the colour combination. This requires the host processor to retrieve the texture indexes from the array, look up the colour value at that index, and return the texture colour back to the SIMD array for combination. This has to be done at the texture index calculation time of each pixel. Since it requires reading and

writing to the SIMD array during the rasterization process, it is relatively inefficient. At each texture index calculation time, it is unknown whether or not the pixel will be in the final image, so the effort in fetching the texel may be completely wasted. This example only describes point sampling on the texture, let alone the much more common bilinear interpolations. With bilinear interpolation several adjacent pixels must be fetched and set to the array.

The second approach is to have the SIMD array store a secondary buffer with the texture indexes for each pixel currently in the frame. This works exactly like the colour and depth buffers, and is masked in the same way as the colour buffer. This means at the end of the frame, when all triangles are drawn there is a complete list of texture coordinates in addition to the colour buffer. Now the host processor fetches all of those texture colours and applies them to the Gouraud colours as it retrieves the colour frame buffer from the Array Processor. This means, unlike the first solution, no texture lookups are wasted and it does not require writing back any data to the array processor. The downside of this approach is that it costs more memory, since an entire additional buffer is required for texture indices.

Figure 4.10: A black and white stripe texture modulated with the Gouraud shading

4.6 Performance

The Atsana Tool Centre simulator provides a whole profiler which breaks down the performance of the SIMD array [17]. The following table shows the break down of each step of the algorithm.

Data	***Clock Cycles***
Triangle Fill	41956
Depth Calculation and Test	13712
Colour calculation	28998
Texture Coordinates	17466
Total	***102132***

Table 4.6: Clock cycle breakdown of algorithm stages

Originally, a third of all clock cycles were the 8bit * 16bit multiplications used for the line equations. Since utilizing the cumulative coefficient interpolation described in the triangle drawing implementation, there are only three multiplications required per triangle. Since the update, 25% of the reduced

cycle count is the 16 bit subtraction. Other significant operations are loads, and stores to registers and the write enable, taking most of the remaining cycles.

The cycle count of this triangle drawing and shading algorithm, excluding host sequential code is 102,132 clock cycles. The array runs at 96 MHz, meaning that these operations can be run 940 times per second, at a maximum of 96 triangles per run. This equates to 90,240 triangles per second, peak performance. At 20 frames per second, a reduced frame rate which is typical for portable devices, it works out to 4,512 triangles per frame.

This performance was compared against a measured tight loop running this algorithm 100 times per measurement on the J2210 hardware. The performance of the algorithm on the hardware is comparable to the simulator.

Platform	***Operations Per Second***
Simulator	940
Hardware	909.09
Error	***3.3%***

Table 4.7: Hardware vs Simulator performance measurement

While a relatively high peak performance is possible, it is heavily dependent on the location of the triangles. For example, if 100 triangles occupy the same computational units' screen space, the entire frame throughput is cut by a factor of 100. This is not an unlikely scenario, either. It is shown when rendering the Newell teapot [18], shown in Figure 4.11. The complex top handle

portion of the tea pot places 107 of 896 triangles inside one computational unit. This cuts the peak throughput of the array to 880 triangles per second, or 1 frame per second. While the teapot is an extreme example, 3D scenes tend to have a heavier concentration of polygons in the center of the screen.

Figure 4.11: The Newell teapot rendered by the SIMD array with texturing disabled

While slow compared to commercially available dedicated high-power desktop 3D processors, the speed is quite impressive considering the architecture. Assuming small triangles, the triangles per frame, or display size can increase linearly with the number of CUs. It is also important to consider the energy consumption of the algorithm on the array processor.

4.7 Power and Energy

By using the hardware implementation of the algorithm the power consumption can be measured. The power measurements are performed by placing an ammeter in series with the 1.2V voltage regulator in order to measure current [19]. This excludes the Input/Output pins which run at a higher 2.5V. The 1.2V regulator is

replaced by a Tektronix CPS250 Triple Output Power Supply, and the input current is measured using a Fluke 187 True RMS Multimeter.

The 1.2V supply gives power only to the J2210 core circuits (not I/O pads), providing direct current consumption for only the J2210, not the other peripherals on the board. The J2210 itself contains the ARM 922T RISC, the Array Processor and supporting Array Controller, and a series of other peripherals for communication. The current is measured through consecutive execution loops, and DC power is calculated.

$$Power = Current \times Voltage \qquad (4.2)$$

With the power consumption and the execution time, the energy consumption can be calculated.

$$Energy = Power \times Time \qquad (4.3)$$

In these tests, all unused peripherals are put in sleep mode. The test cases are as follows.

Test Case A – Leakage Current

The J2210 CPU is shut down via the power down input chip on the CEB. This shuts down the DLL and clock inputs.

Test Case B – ARM922T™ Full Screen Sequential Algorithm

The ARM922T™ is executing its sequential rasterization algorithm on a full screen rectangle, effectively measuring the peak fill rate of the sequential algorithm.

Test Case C – Array Processor Full Screen SIMD Algorithm, AP Halted

The Array Processor is executing the SIMD rasterization algorithm on a full screen rectangle, effectively measuring the peak fill rate of the SIMD algorithm. The Array Processor is disabled to measure the consumption of the Array Controller and Computational Memory (CMEM) Interface Unit (CIU), which handles communication between the Array Processor and ARM.

Test Case D – Array Processor Full Screen SIMD Algorithm, AP Active

The Array Processor is executing the SIMD rasterization algorithm on a full screen rectangle, effectively measuring the peak fill rate of the SIMD algorithm. The Array Processor is enabled, and producing proper output.

Test Case E – ARM922T™ Teapot Sequential Algorithm

The ARM922T™ is executing its sequential rasterization algorithm on the teapot model in order to characterize a more realistic example. The teapot contains roughly 1000 polygons.

Test Case F – Array Processor Teapot Sequential Algorithm, AP Active

The Array Processor is executing its sequential rasterization algorithm on the teapot model.

The current and time measurement of each test is shown below. The texture coordinates are calculated but not applied in order to calculate a fair comparison between both rasterizers.

Test Case	*Current*	*Processing Time*
A Leakage	3 mA	N/A
B ARM FS	47.9 mA	23 ms
C AC Only	79.5 mA	N/A
D AP FS	107.2 mA	2.2 ms
E ARM Tea	52.3 mA	610 ms
F AP Tea	105.5 mA	780 ms

Table 4.8: Current consumption of the test cases

By using the current measurement it is possible to compare the power consumption per device in the full screen power test.

ARM 922T	*AC + CIU*	*Array Processor*
57.5 mW	37.92 mW	33.24 mW

Table 4.9: Power consumption broken down by hardware

It is clear that while the Array Processor takes relatively little power compared to the ARM, the Array Controller and CMEM Interface Unit do add more than 100% overhead in terms of power. The energy consumption is entirely dependent on the execution time of the algorithms.

Test Case	*ARM922T™ Energy*	*Array Processor Energy*	*Energy Ratio ARM to AP*
Full Screen	1.3 mJ	0.155 mJ	8.4×
Teapot	35.0 mJ	55.5 mJ	0.63×

Table 4.10: Energy consumption

While the full screen rendering shows a significant performance increase of 10.5× and 8.4× less energy, it is an easy problem with peak parallelism. Despite the peak performance of the SIMD algorithm being significantly faster, the more realistic example shows that the weakness of the tile based algorithm is apparent.

On the other end of the spectrum, the teapot puts more than 10% of its triangles inside one CU. Since the bottleneck exists in one CU, the others are executing needlessly. The fact that the teapot contains many polygons which are physically close, means wasted work on most of the processing array, which means wasted energy. The teapot problem shows a slightly longer processing time and 1.5× more energy.

The actual energy consumption is sensitive to data for this algorithm, given an unknown amount of processing time for any scene. Since 3D scenes generally contain more polygons in the center, this bottleneck is likely.

4.8 Memory

Memory consumption inside the array is important, due to limited space per CU. While the architecture would allow for more memory per CU, the SIMD array

takes up significant die space on the chip. The following table shows the breakdown of memory used for this algorithm.

Data	*Number of values*	*Data Size*	*Total Consumption*
Line Coefficients and other constants	96	2 Bytes (short integers)	192 Bytes
Colour value	8×36 = 288 (number of pixels per CU)	2 Bytes	576 Bytes
Depth value	8×36 = 288	2 Bytes	576 Bytes
Mask/Write Enable	8×36 = 288	1 Byte	288 Bytes
Texture Coordinates	8×36 = 288	4 Bytes	1152 Bytes
Total			***2784 Bytes***

Table 4.11: Memory Usage

This leaves 1312 Bytes free in each CU. The maximum number of pixels becomes 432 per CU, which could be arranged as 12×36 or 9×48 per CU, or a complete frame buffer size of 288×144 or 216×192.

4.9 Summary

The tile based rendering algorithm is realizable on the J2210's Array Processor. The algorithm was fully implemented and functional. While performance numbers for this algorithm can be very positive, the unpredictable nature of input can cause significant slowdowns for complex scenes, specifically where many overlapping polygons exist.

The performance variance directly translates to energy consumption, which can show results as high as 8.4× lower energy consumption, but can also be

shown to be negative in more realistic tests. The tile based algorithm has been implemented and proven on the hardware, but the results are data dependent.

Chapter 5
Pixel Based Rendering Approach

In this chapter we examine rendering using SIMD parallelism over pixel space. The tile based approach, examined in the previous chapter, is effective in certain cases, but performance is unpredictable in a real application. The biggest problem with the tiled approach is the gross inefficiency when dealing with small triangles in leading to a poor ratio of useful to non-useful work. Our second SIMD parallelization approach handles processing on a much finer grain parallelism, each CU dealing with one pixel at a time. This will result in 96 pixels of arbitrary screen coordinated processed simultaneously. No calculations will be performed on behalf of a pixel which is outside a triangle. We will show that the pixel based processing algorithm can yield a performance increase of 5× while consuming 7.7× less energy on small patches.

While vertex processing is an inherently parallel operation, the pixel operations can take significant advantage of sequential computations. Each pixel

can be processed faster using the results of the previous pixel in a common triangle. The method of this approach is using the gradient slopes and summing with the corresponding value of the previous pixel.

$$R_{x,y} = R_{x-1,y} + \frac{dR}{dx} \tag{5.1}$$

This approach is appealing because the slope, *dR/dx* is always required and known, and the operation requires only an addition. Of course any arbitrary point can be calculated using a less optimized equation, which uses much more expensive multiplication.

$$R_{xy} = R_0 + x\frac{dR}{dx} + y\frac{dR}{dy} \tag{5.2}$$

This equation holds true for any pixel in the triangle, based off of a common starting point. While both provide the same result, barring rounding error, the latter requires four times as many ALU operations, including possibly slower multiplication operations. So while a sequential processor always has the previous data on hand, this data will be absent to the parallel processor, since two adjacent pixels may be operated on simultaneously. This means the SIMD processor will have to overcome the performance hit of a greater number of arithmetic operations. Also, we still show that the existing communication methods between the host and Array Processor pose a bottleneck to effective performance.

5.1 Host/Array Division of Work

In this approach the ARM host will test each pixel for triangle inclusion prior to sending it to the SIMD array. The ARM will operate on a triangle by triangle basis, breaking each down into a number of included pixels, and then sending the result to the SIMD array. While optional, it may also be desirable for the ARM to do a pre-emptive depth calculation, test and write, for a few reasons. First, the ARM has quick access to the necessary information in the depth buffer. Second, the precision of a 32 bit depth is important to prevent z-fighting issues. Depending on the application, floating point depth may be necessary. This also prevents the SIMD array from processing pixels that would be thrown out for failing a depth test, but does require more work from the ARM. The parallel execution of work is shown in Figure 5.1.

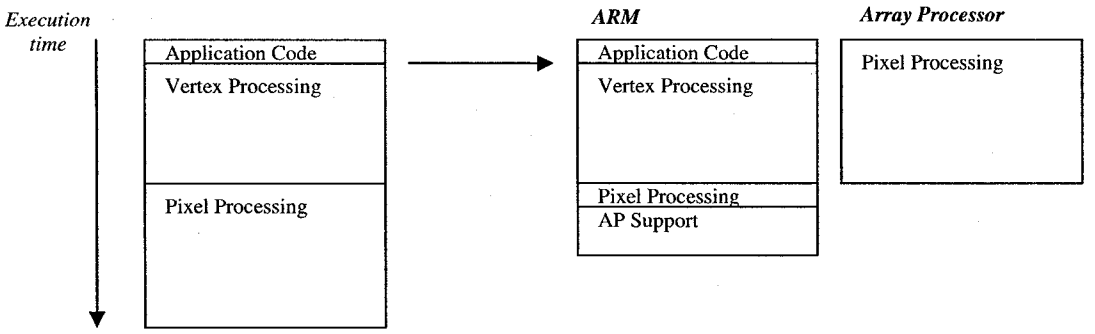

Figure 5.1: Parallel Execution Comparison

The J2210 array processor is designed for small, low power systems, and has computational power suitable for integer arithmetic and logic operations. It is

interesting to see whether a programmable SIMD array can be used in traditional patch processing in real time 3D rendering engines. Patch processing, including colour, shading, depth calculation and texture mapping, is analyzed using the computational units that make up the array processor in the Atsana J2210. SIMD processors that are tightly coupled with memory can perform memory operations required for graphics efficiently [20]. A host processor can be used to perform the floating point vertex level calculations and transfer the patches into the array processor for pixel level interpolation operations.

Re-examining the pixel processing, this implementation slices the division of work slightly earlier than the tile based approach, pulling triangle drawing out of the array processors responsibility. The division of work is shown in Figure 5.2.

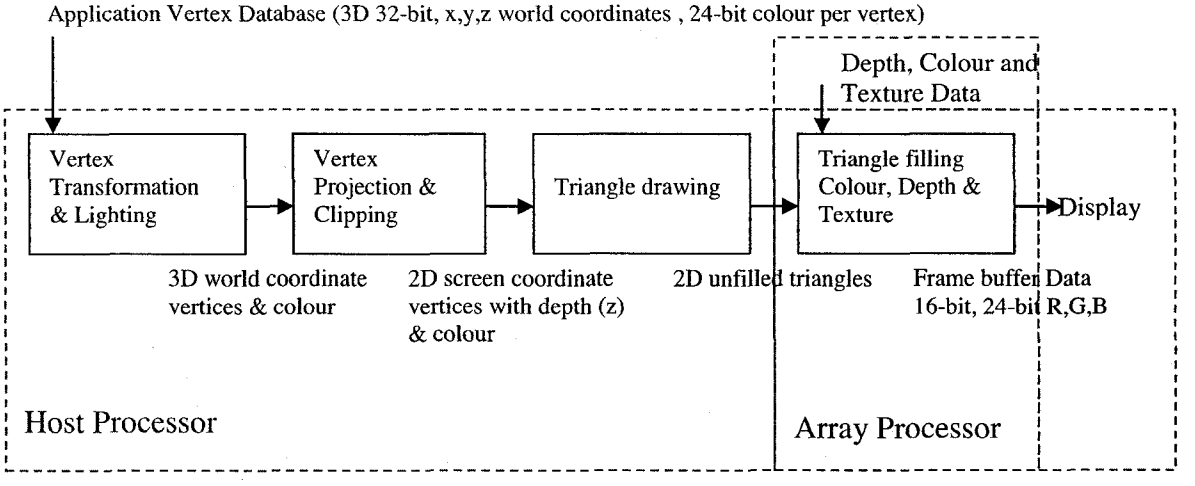

Figure 5.2: Processor division of work

By removing the triangle drawing from the array processor we reduce the wasted work from the tile based approach, caused by always working on every pixel, whether it is in a triangle or not. The drawback of removing that responsibility is the extra work for the host processor. While the boundaries on processing look relatively clear cut, the array processor is not strictly independent in the triangle filling stage. Since the colour and depth buffers are kept outside of the array processor, it requires information from the host at specific intervals. These transactions are described in the sections that follow.

5.2 Host/Array Communication Implementation

Each vertex in the scene will become part of one or many polygons, which have to be drawn to create the scene, called patches. At this point the vertices are projected into screen space, and have to be filled with pixels in order to represent the triangles on a 2D frame buffer.

The triangle processing loop is shown below:

```
for each triangle
    calculate line equations ax + by + c = 0
    calculate colour interpolation equations
    calculate depth interpolation equation
    calculate texture interpolation equations
    for each scanline in bounding box
        for each pixel in scanline
            test pixel for inclusion
            if inside triangle
                send for pixel processing
```

Table 5.1: Pseudo code for triangle inclusion test

In a simple case, each vertex is tagged with 7 pieces of information. First, four pieces of colour information, red(R), green(G), blue(B), and alpha transparency(A). These values can be specified by the 3D model data, but are more likely a result of coloured lighting calculations done in vertex processing. Then, for each texture applied to the patch there is a horizontal(S) and vertical(T) index into that texture at each vertex. Finally each vertex is tagged with a depth (Z) value. Since this information is only provided at the vertices, some level of interpolation is required for the other pixels.

The pixel processing loop is shown below:

```
for each pixel
    fetch currentZ
    interpolate Z
    if Z < currentZ
        interpolate R
        interpolate G
        interpolate B
        interpolate A
        get current framebuffer colour
        combine R,G,B,A, current colour
        interpolate texture coordinate S
        interpolate texture coordinate T
        fetch texel S,T
        blend texel texel, colour
        write pixel to framebuffer
        write depth value to Z buffer
```

Table 5.2: Pseudo code for pixel processing

Each of the iterations depends only on the current pixel data being operated on, as well as the current contents of the colour buffer and Z buffer at that coordinate. The only concern for doing the operations in parallel is to ensure that two pixels of the same x and y coordinates are not being operated on simultaneously. If this is the case the depth buffer will ensure only the correct pixel data gets written.

Once each vertex is projected to screen space by the host processor a significant amount of data must be sent to the array processor for pixel computation. For each attribute interpolated there are three values, the initial value, usually starting from the top of the triangle, and the change in that value with respect to x and y as it traverses the triangle. Multiply these three values by the 7 attributes (R, G, B, A, S, T, Z) for 21 values to be calculated. Additionally, since each computational unit is handling an arbitrary pixel, it needs to know its relative X and Y position in the 2D frame buffer. A total of 23 values need to be written to each CU, which is a very expensive operation. However, 21 of those values are common for each CU if the entire array process is working on the same triangle, leaving only the X and Y location of the pixel to be unique for each CU.

If each of 96 CUs is processing a pixel within the triangle, requiring 23 values, then a total of 2208 (23 values for 96 CUs) 16-bit integers must be written to the array processor per triangle. The latency of writes into the array processor memory creates an impractical amount time spent strictly feeding data to the array

processor. The measured time for this much data is listed in the result section below.

While this particular approach may not be appropriate to the J2210 due to significant data writes, calculating the actual computational work is significant and useful. While 2208 writes is not suitable it is important to remember that each CU only has 2 unique values, its X and Y coordinate. Which means the other data, 21 words times 96 CUs, is shared. Since the array processor features a processor in memory architecture, it would be possible to enhance the hardware to support a broadcast write, which wrote these shared values directly into the values of all the CUs. This enhancement is discussed further in chapter 6. If added the number of writes would be reduced to the 21 interpolation values, as well as the X and Y coordinates for each CU. This brings input required down to a more manageable 213 (21 attribute values plus 2 values per CU) values, or 2.2 values per computed pixel.

5.3 Triangle Drawing

As described above, the host processor will do the actual drawing of the triangles, delegating the actual triangle filling to the array processor. The host processor uses a triangle scan conversion algorithm to determine each pixel's inclusion in the triangle based on the three line equations that describe it. It starts at the top point of the triangle and steps from the beginning of the precalculated edge until it

runs into the second bounding edge. Upon reaching the second edge it moves down to the next line in the triangle and repeats the process until the last line. This process is shown visually in Figure 5.3.

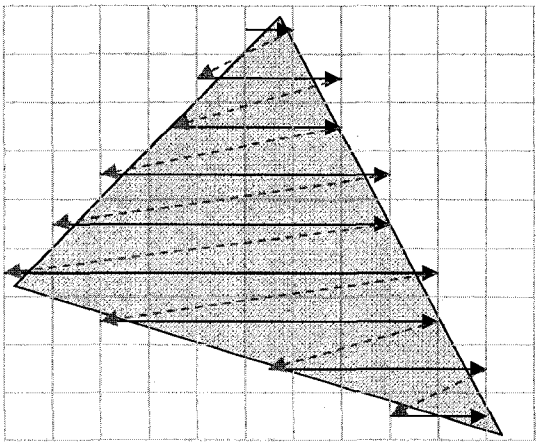

Figure 5.3: Scanline processing algorithm implemented in the ARM

While this is run by the host processor, it still uses the same fixed point integer arithmetic that is used by the array processor. The fixed point implementation is useful not only for speed, but for comparison of results between the two platforms, in terms of performance and the correctness of both algorithms.

Once each pixel is determined to be inside the triangle, the X and Y values are stored in a data structure to be sent to the array processor for computation.

5.4 Gouraud Shading Implementation

This method requires using the computational units as general purpose pixel processors. The CUs themselves are not associated with where the pixel lies. They are assigned a set of interpolation values and x and y coordinates to process. The CUs can be treated as a pooled resource of data, and balanced by the host processor.

This calculation of a single pixel value is merely an evaluation of the arbitrary point equation shown below:

$$C_{xy} = C_0 + x\frac{dC}{dx} + y\frac{dC}{dy} \tag{5.3}$$

C, in this equation, is any of the colours for this coordinate. The same equation calculates the red, blue, green and alpha values. All the inputs are sent to the array processor prior to computation, so the result can be evaluated immediately. Figure 5.4 shows the division of pixels over CUs for one example triangle.

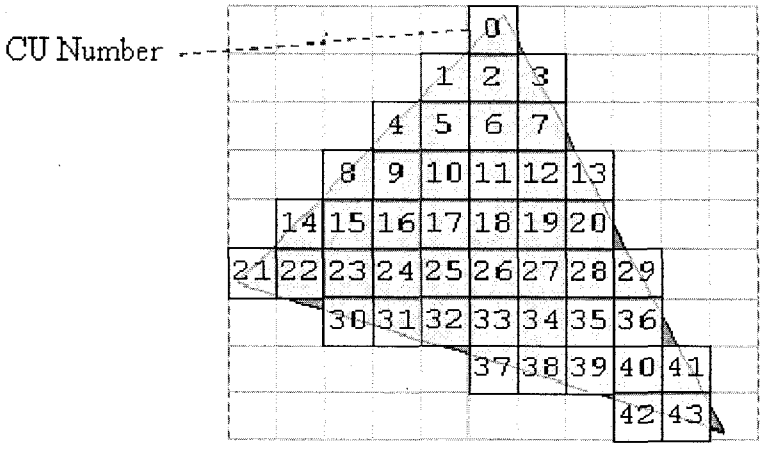

Figure 5.4: Division of work for single pixel per CU. 43 CUs are used to fill the triangle

The drawback of this equation is the two multiplication components which require more than 10 times the cycles required for an addition. For large triangles it is faster to have each CU calculate a few, say four, adjacent pixels, and increase the granularity of the input. For each of the three neighbouring pixels, only an addition is required, as shown below.

$$C_{x,y} = C_{x-1.y} + \frac{dC}{dx} \tag{5.4}$$

Sending every individual pixel to a CU may be appropriate in some cases, but in others sending every second pixel horizontally and vertically to the CUs will yield much faster results, since the addition required to compute a neighbour is much faster than the multiplication for an arbitrary pixel. While there will be some wasted work on pixels outside the triangle, the cost is low. Each extra pixel

addition adds roughly 6 extra clock cycles per attribute, including the load and store. Over 6 attributes and 3 extra pixel additions it works out to roughly 108 cycles, on top of the 1248 cycles for calculating the original pixel, or 8.6% more work for 4 times the results. This does not actually create much work for the host processor, since it is already aware of which pixels are inside and which are not, and will grab them accordingly from the array processor. We implemented this logic in the Array Processor algorithm, but have not implemented the logic for parsing and removing unused pixels on the host side. An example is shown in Figure 5.5

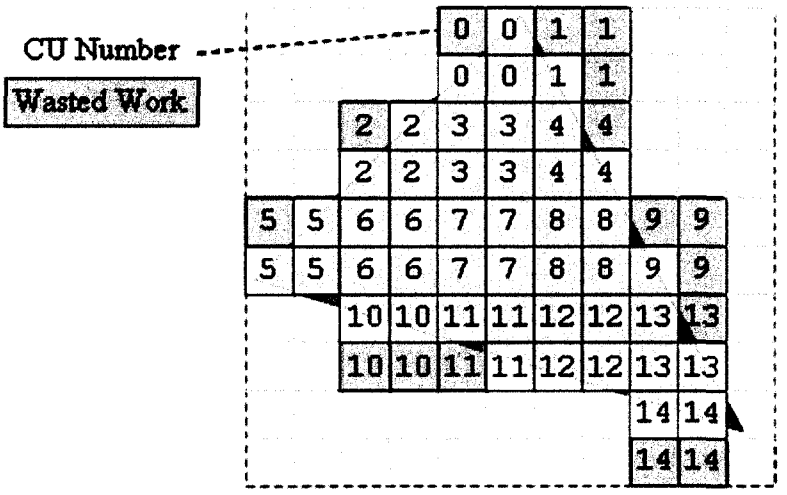

Figure 5.5: Division of work for four adjacent pixels per CU, 14 CUs are used

5.5 Depth(Z) Calculation and Evaluation

The depth value can be calculated in the same way as the colour values above. In fact from the array processor's perspective the exact same program can be run and produce the correct results.

$$Z_{xy} = Z_0 + x\frac{dZ}{dx} + y\frac{dZ}{dy} \tag{5.5}$$

However, in this case the depth calculation may be more appropriate outside of the array processor. Since, unlike the tile based method, the depth buffer is stored outside of the array processor there is little advantage to having the value computed inside the array processor. In fact since the depth test can immediately determine if a pixel should be processed, having the array processor do work on a masked pixel is entirely wasted work.

If the host processor does the depth test on a per pixel basis before sending the result to the CU it can choose to avoid the unnecessary work altogether at little cost. Simply adding the depth test to the triangle inclusion test described above will eliminate processing time spent on obscured pixels, leaving the array processor open for unmasked pixels as shown in Figure 5.6. If the entire triangle is found to be obscured before rendering the pixel programs do not even need to be run.

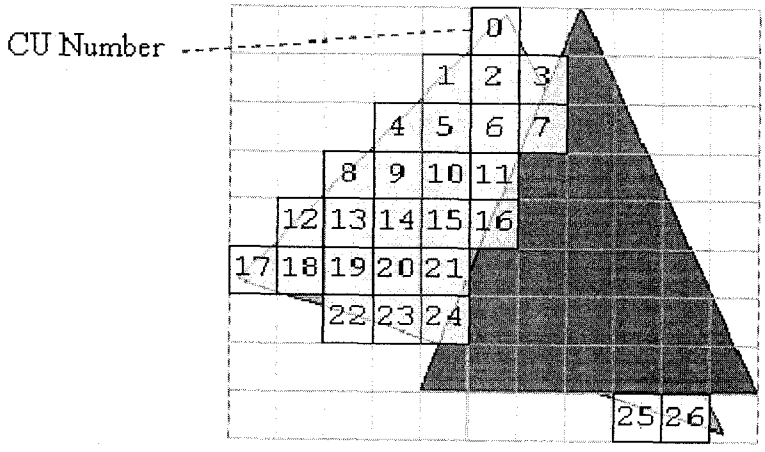

Figure 5.6: A triangle obscures the first triangle, reducing the number of pixels sent to the AP

5.6 Texture Coordinate Calculation

The texture coordinate equations can be calculated as the Gouraud colours above, using either the arbitrary point calculation, or the combination of arbitrary point and small tile.

$$S_{xy} = S_0 + x\frac{dS}{dx} + y\frac{dS}{dy} \tag{5.6}$$

$$T_{xy} = T_0 + x\frac{dT}{dx} + y\frac{dT}{dy} \tag{5.7}$$

Once calculated however, these values are merely indexes into a texture colour array that is completely unknown to the array processor. It is impossible to store the textures inside each CU, not feasible to index them, and impractical to

store them all across the array processor, so they must remain inside system memory.

This requires a two way communication between the host and each CU at this stage in the process. Upon completion of the texture coordinate program, the host must grab all of the texture indices from the CUs and convert them to the memory address where the texture is stored.

```
for each CU
    read S and T
    convert S and T to texture memory address
    read address data
    write data back into CU
```

The amount of data going into and out of the array processor is not small, 23 values per pixel in, and 7 out. Not only is the data movement costly, but this completion will stall the array processor until it can retrieve the data, which means the host processor will consistently be interrupted in order to keep the array processor busy. The data movement and the context switch for the host processor are both costly, hurting the feasibility of arbitrary texturing on the array processor side without specialized hardware. Some suggestions for this deficiency are addressed in chapter 6.

Once the texture value is collected the value is modulated with the Gouraud shaded value to form the output colour. If alpha blending is enabled, the output colour is blended with the previous frame buffer value provided with the

initial data. At this stage the resulting image data is retrieved and placed in the frame buffer in system memory, and the process is complete.

5.7 Performance

To evaluate the performance of the array processor, it must be compared with a suitable platform. Currently, most portable devices use a RISC processor for software graphics processing, like the ARM found in the J2210. The array processor is clocked at 96 MHz with the ARM 922T at 192MHz, computational performance of the pixel algorithms are measured. The following algorithms are measured from test cases on the CEB using controlled test cases. The test cases perform the algorithm in a tight loop overall several seconds to measure the result. The shade algorithm is described above, and calculates any of the specific R,G,B,S,T,Z,A values at individual coordinates within the triangle. The results are shown in Table 5.1 in millions of operations per second.

Algorithm	*ARM 922T*	*Array Processor*
Shade	1.41	0.077

Table 5.3: Algorithms in millions of operations per second

While the array processor is significantly slower in both operations, with each pass of the algorithm it produces 96 results. Normalizing the results to 96 CUs shows a generous increase in performance.

Algorithm	*ARM 922T*	*Array Processor (per result)*
Shade	1.41	7.39

Table 5.4: Algorithms in millions of operations per second, throughput

In processing single pixel patches, the array processor can exceed the speed shading algorithm by more than a factor of 5. However, this result is also dependent on two factors. The first is that all 96 processors are busy in order to receive the maximum efficiency. Since scenes will be generated from millions of pixels, this should not be an issue. The second is that the patches are single pixels in size. In practice, the sequential processor can exploit values of neighbouring pixels to reduce computational cost. As performed in the tile algorithm, the sequential CPU can perform a small delta addition when calculating the nearest neighbour pixel seen in equation 5.1, instead of the full calculation shown in 5.2. This means that the CPU can replace the full shading algorithm with addition in larger patches. As patch size increases the array processor loses its edge. The following shows the processing time comparison per attribute for patches of increasing size.

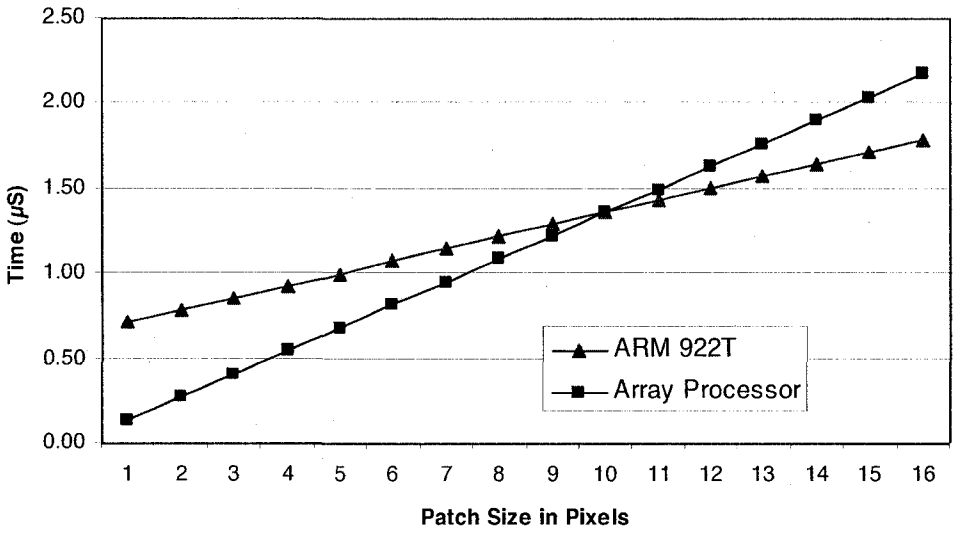

Figure 5.7: Processing time comparison for varying patch sizes

As shown in Figure 5.7 the array processor starts out significantly faster for smaller points. As it scales up however, the ARM is able to exploit the small marginal cost to interpolate via delta additions instead of recalculating the gradients with each pixel. The performance of the ARM surpasses the array processor as the number of pixels per patch increases. The break-even point is at a 10 pixel patch size.

The AP can also be programmed to exploit the same nearest neighbour interpolation by having a fixed set of coordinates to operate on. Since the Array Processor CUs all have to execute the same algorithm, the nearest neighbours must be calculated by all CUs, needed or not. As described above, calculating 3 neighbouring pixels only incurs an 8.6% penalty, so by programming the AP to calculate 4 pixels at a time, it can calculate all 4 at nearly the same cost as a single

pixel. Since all 4 pixels will not always be required, some calculation time may be wasted; however, the wasted time is almost negligible since it adds very little computation time. This brings the AP ahead of the performance of ARM for larger patches as shown in Figure 5.8.

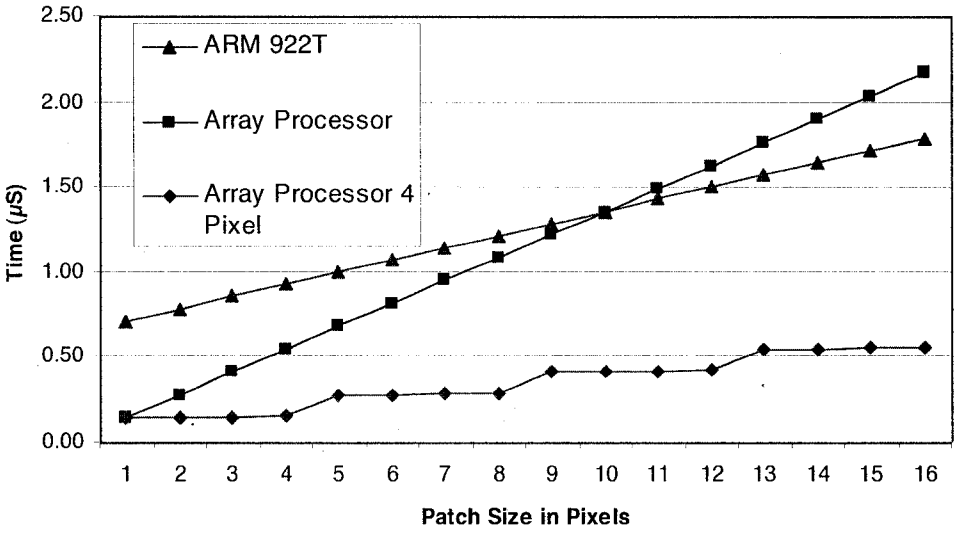

Figure 5.8: Processing time comparison for varying patch sizes including the 4 pixel array processor neighbour calculation

All the calculated pixels will not be used, so work will be thrown away with this method, but even at 50% efficiency, say only 2 of every 4 pixels calculated are used, the algorithm still remains under the ARM in terms of processing time.

At a 2.9% increase in processing time per pixel added, it is important to calculate the ideal number of adjacent pixels to calculate for varying patch sizes.

It is difficult to estimate the ideal block size for each average patch size due to the fact that the patches can vary from extremely orthogonal and having even dimensions to extremely thin and angled. The following table shows the number of blocks of each size to cover an isosceles right angle triangle of varying patch sizes, orthogonal to the screen. The patch size increases by the number to create the next right angle orthogonal triangle.

Block Size	***Patch Size (pixels)***						***Cycles Per Pass***
	1	***3***	***6***	***10***	***15***	***21***	
4 (2×2)	1	1	3	3	6	6	1356
9 (3×3)	1	1	1	3	3	3	1536
16 (4×4)	1	1	1	1	2	3	1788
25 (5×5)	1	1	1	1	1	3	2112

Table 5.5: Number of blocks required to cover an orthogonal isosceles triangle of varying size

Multiplying out the number of passes by the number of clock cycles per block calculation we can see the fastest in each size.

Block Size	***Patch Size (pixels)***					
	1	***3***	***6***	***10***	***15***	***21***
4 (2×2)	***1356***	***1356***	4068	4068	8136	8136
9 (3×3)	1536	1536	***1536***	4608	4608	***4608***
16 (4×4)	1788	1788	1788	***1788***	3576	5364
25 (5×5)	2112	2112	2112	2112	***2112***	6336

Table 5.6: Number of cycles to calculate all the pixels in orthogonal isosceles triangle, based on varying block sizes

The following table shows the number of blocks of each size to cover the worst case patch shape for a square calculation block, which is a line.

Block Size	*Patch Size (pixels)*						*Cycles Per Pass*
	1	*3*	*6*	*10*	*15*	*21*	
4 (2×2)	1	2	3	5	8	11	1356
9 (3×3)	1	1	2	4	5	7	1536
16 (4×4)	1	1	2	3	4	6	1788
25 (5×5)	1	1	2	2	3	5	2112

Table 5.7: Number of blocks required to cover a line of varying size

Again we can find the fastest block in each patch size by multiplying by the number of cycles per block calculation.

Block Size	*Patch Size (pixels)*						*Cycles Per Pass*
	1	*3*	*6*	*10*	*15*	*21*	
4 (2×2)	***1356***	2712	4068	6780	10848	14916	1356
9 (3×3)	1536	***1536***	***3072***	6144	7680	10752	1536
16 (4×4)	1788	1788	3576	5364	7152	10728	1788
25 (5×5)	2112	2112	4224	***4224***	***6336***	***10560***	2112

Table 5.8: Number of cycles to calculate all the pixels in a line, based on varying block sizes

Since we are dealing with a small display size, the average patch size will be low. The 2×2 and 3×3 blocks perform best on 1-10 pixel patches but even the 5×5 block performs well on smaller patches. Again, the best size is dependent on the input data.

5.8 Power and Energy

The energy consumption of both algorithms is measured using the methods described in chapter 4. These results are for a small patch size of 1-4 pixels, where the SIMD processor may be considered to have a strong advantage.

However, as shown in the performance section the linear increase of the ARM processing time per pixel can be nearly matched by the Array Processor. Again the results can be broken down by the 3 major units of the processor.

Test Case A – ARM922T™ Shading Algorithm

The ARM922T™ is executing its sequential shading algorithm for attribute calculation in a tight loop. Processing time is shown for 192,000 calculations.

Test Case B – ARM922T™ Idle

The ARM922T™ is idle, simulating consumption while waiting for the array processor to complete. It is not placed in sleep mode, but fed a series of NOPs.

Test Case C – Array Processor Shading Algorithm, AP Halted

The Array Processor is executing the shading algorithm for attribute calculation in a tight loop. The Array Controller and CIU are active, but the Array Processor is in sleep mode. Processing time is shown for 192,000 calculations.

Test Case D – Array Processor Shading Algorithm, AP Active

The Array Processor is executing the shading algorithm for attribute calculation in a tight loop. The Array Controller, CIU and Array Processor are active. Processing time is shown for 192,000 calculations.

Test Case	*Current*	*Processing Time*
A ARM Shade	52.3 mA	136 ms
B ARM Wait	43.5 mA	N/A
C AC Only	54.3 mA	N/A
D AP Shade	78.7 mA	26 ms

Table 5.9: Current consumption of test cases

The power consumption of the test cases is calculated from the current. The ARM numbers are from test case A, the Array Processor numbers are based off of cases B and C while subtracting the idle consumption of the ARM from test case A.

ARM 922T	*AC + CIU*	*Array Processor*
62.76 mW	12.96 mW	29.28 mW

Table 5.10: Power consumption broken down by hardware

It is notable that the power consumption of this algorithm when running in a tight sequence is significantly lower than the tile based algorithm for the Array Processor and hardware, which can be explained by the instructions. The algorithm provides fewer instructions that take longer to execute with multiplications, instead of additions, so there is less instruction fetches. This algorithm also works much more out of register arithmetic than the memory intensive tile based algorithm, meaning fewer loads and stores.

Energy consumption is calculated by using the measured execution time and power. On single pixel patches the performance of the SIMD algorithm is faster by a factor of 5× and lower on energy consumption by nearly 8×.

Test Case	*ARM922T™ Energy*	*Array Processor Energy*	*Ratio (Arm to AP)*
C&D	8.5 mJ	1.1mJ	7.7×

Table 5.11: Energy consumption of 192,000 single pixel calculations

As shown in the performance section, this is the ideal case for the Array Processor, however even the larger patch sizes can be completed in less time using the Array Processor as shown in Figure 5.8. Performance aside the power consumption of this algorithm on the Array Processor is 2× less than the ARM922T™ when active assuming both are running constantly. As the number of pixels per patch increases, the energy consumption for both the ARM and AP increase linearly, but since the AP uses less energy the break-even point is at a patch size of 34. The energy consumption for each algorithm is shown in Figure 5.9.

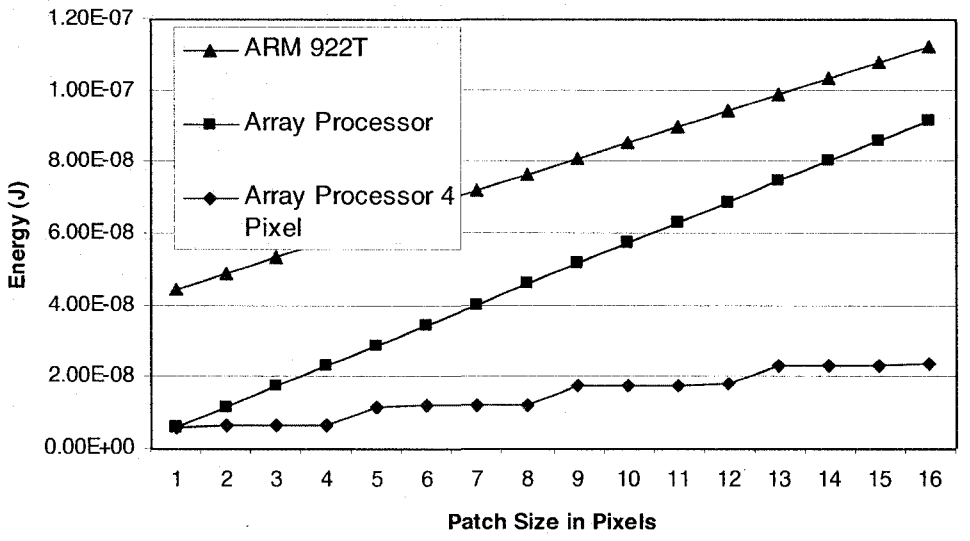

Figure 5.9: Energy comparison for varying patch sizes including the 4 pixel array processor neighbour calculation

5.9 Memory

The AP seems ideal for pixel processing on small patches, but it is important to note the importance of access to external memory. External accesses are required at the beginning of the algorithm for the current depth and colour values for the depth test and colour blending. These can be provided by the vertex processor at the beginning of the algorithm. Texture data is more complicated; the processor does not know what texture data is required until it calculates the texture coordinates. At that point the texture data at the calculated address will need to be fetched.

The texture coordinates need to be translated into absolute memory addresses by the host processor, then the data must be fetched and given back to the array processor.

While the array processor can be shown to outperform the ARM in computational power, it is important to consider its relationship to the system memory. To use a SIMD array processor a specialized interface to memory could be implemented as a coprocessor. A coprocessor with rapid access to depth buffer, colour buffer and texture data would allow the AP access to all required data without interfering with the host processor.

5.10 Summary

The array processor is suitable for pixel calculations, and performs well both in terms of performance and especially energy with up to 5× faster performance consuming 7.7× less energy. However, it is lacking in some areas which make it an incomplete solution without data serving from the ARM host. It needs arbitrary access to the colour and depth buffers, as well as texture memory. A DMA interface between the SIMD array and the memory system would provide enough capability to allow the array processor independence from the ARM for pixel processing.

Computational performance of a SIMD architecture for pixel processing was shown to exceed that of a sequential RISC processor, especially as the size of

the polygons decreases. The low energy requirement would make it ideal in an embedded application. In addition to computational performance however, pixel processors need fast access to the data that they are manipulating. Sequential processors can take advantage of cache coherence to reduce the memory latency and bandwidth requirement.

A SIMD processing array tied tightly to a custom DMA engine would prove a beneficial solution for accessing and processing pixel data. A cache would reduce latency in texture mapping and buffer writes. The parallelism inherent in pixel processing makes a SIMD architecture quite suited to the computations required. With the correct memory interface, a SIMD processor makes a powerful, appropriate, solution for real time 3D rendering.

Chapter 6

System Level Communications and Efficiency

SIMD to Sequential communications can become a bottleneck due to the amount of data to keep a SIMD array full. In cases where the SIMD system is not the sole processing unit, it is important to establish system level parallelism. The J2210 uses the ARM922T™ processor as the host. It provides data and calls high level instructions for the Array Processor, but it is also free to work on problems itself, in parallel with the Array Processor. This requires a level of synchronization. In the J2210 the array processor is the back end processor or slave. The only control it has over the ARM is the ability to issue an interrupt when idle. System performance can increase dramatically if both processors can be kept fully utilized. Of course this ideal is not attainable in practice, but steps can be made to obtaining high utilization. We will describe several communications issues found in the J2210 and comment on possible solutions.

6.1 System Level Parallelism

For ideal parallelism, neither processor should be blocked waiting for the other. In this case, since the array processor requires instructions and data from the host, this is not easily maintained. While the J2210 does allow the Array Processor and the ARM to execute in parallel, constant communication is required for them to form an efficient system. As vertex data is being processed by the ARM, ideally the pixels corresponding to the previous set of vertex data should be processed by the Array Processor. This is possible with the tile based implementation because the host and array responsibilities are much more defined.

If texturing is disabled the Array Processor handles everything after the projection stage, allowing itself to complete the final image in its own frame memory. At this point the only work for the ARM is writing out the data to the display device. If texturing is enabled, then the division of work becomes less clear. As the data is being pulled out of the processor the texture data must be applied, since the Array Processor has no ability to fetch data from system memory directly. This interruption slows the flow of data, but both processors can still act quite independently in parallel.

In the pixel based approach discussed in Chapter 5, there is much more host babysitting involved. Since the Array Processor is effectively used as a low level pixel processor the data inputs and outputs must constantly be fed and

retrieved. This two way communication ended up becoming a bottleneck for this algorithm on the J2210. The host ends up with little to no time for vertex processing. Data movement becomes the biggest bottleneck of the system.

6.2 Data Movement

In both algorithms there is an undesirable amount of data communication between the SIMD and sequential processors for them to be used in a real world application. The direct read and write of data from system memory into the array processor memory proves to be a bottleneck.

6.2.2 Host processor Read/Write

The following are the measured results of reading and writing data respectively, using a 32 bit word across 4 CUs. In addition the time results for addition and multiplication using the array processor are listed. Note that one pass of these algorithms should produce 96 results in parallel. The throughput results are shown in parentheses.

Operation	***Time (ns)***
32 Bit Read Operation	830
32 Bit Write Operation	488
16 bit by 8 bit Multiplication	1093.44 (11.39)
16 bit Addition	64.0 (0.667)

Table 6.1: Speed of read and write operations from ARM to AP

The array processor runs at 96MHz, or 10.42 ns per cycle. The ARM 922T is running at 192MHz, or 5.21 ns per cycle. This means a 32 bit write operation takes nearly 100 cycles to calculate the address and write the data into array processor memory. Even this write only provides a single word of data to 4 Computational units, at 8 bits each. In order to fill up the 192 inputs for the 16 bit array processor addition test, it would take 96 of these write operations. In the time of just a single write operation, the array processor could have done 7 parallel additions, producing 672 results. Reading from the array processor is worse, since it must stall the ARM from the beginning of the read, while waiting.

6.2.2 J2210 DMA

The alternative to the direct host involvement in writing is to use Direct Memory Access (DMA) hardware, which would free the host from monitoring the reads and writes. While the J2210 architecture does feature DMA hardware, it is not designed for such small scale transfers for 3D rendering. The DMAs provided are directly connected to the Sensor and Variable Length Coder and not appropriate for data transfer.

6.3 Potential Hardware Improvements for 3D Rendering

While the array processor itself is quite proficient at dealing with the computations required for 3D rendering, the J2210 system as a whole has some communication bottlenecks which break down the performance.

6.3.1 System/Array DMA

A configurable DMA in order to pass data back and forth between system memory and the array processor would alleviate the host processors stalls on reads and writes. The data would have to be formatted in a way such that it was appropriately laid out in system memory, or could be reshuffled by the DMA itself in order to be array processor ready. Ideally since the DMA would run on the system clock, there should not be any issues with clock boundary crossing.

In the tile case, the DMA could be used to pull the entire frame buffer from the array processor, while the host works on the next frame input data. The current system required the host to remove all the frame data from the Array Processor and placed in system memory before it can be provided to the display output port. This operation was measured to take 14ms of host CPU time, and could be handled passively by the DMA in the background.

A DMA could assist in the pixel calculation case as well. While this would free up the host processor from reads and writes, its true potential could be realized in keeping the array processor working as much as possible, providing

inputs and removing outputs. Considering the ratio of data transfer to useful work described in section 6.2.2, the data transfer still may be fast enough to keep the array processor working if expensive multiplication operations are involved. This system may however require relatively complex hardware in order to be configurable enough to be aware of the array processor's idle state, and the location and amount of data currently available to be provided and removed from the AP.

In several cases a broadcast write would be useful, where the same data is written to every computational unit, instead of writing the same information to each CU group. Since the data lines are shared between CUs it should be possible in hardware to unmask all the CU groups while broadcasting the data.

6.3.2 Display output DMA

For use in a real time system, a frame rate should generally be higher than 15 frames per second, ideally in the range of 30. A 15 frame per second rate, allows 66ms of computation time. Measured results of writing data from the system memory frame buffer to the display port takes 36ms. This operation is far too costly for the host processor, and can be done passively by a DMA device transferring data directly from the system memory to the display output port. This may require double buffering, which is not uncommon in real time rendering systems. In double buffering, the renderer alternates between two frame buffers to write to, while the alternate buffer is being read by the display hardware.

6.3.3 Optimal CU and memory configurations

For 3D rendering applications the memory required is less than the sheer amount of space required for the video and image encoding that the J2210 Array processor was designed for. Both the tile based and pixel based algorithms are scaleable in terms of CU usage. While more CUs is beneficial to both, again the issue of data movement overhead described in previous sections hinders the improvement.

If the data issues are resolved, both algorithms are CPU bound and not memory bound, so dividing the memory into smaller areas with more processors would be linearly beneficial to both algorithms.

6.4 Summary

Communication is essential for proper system level parallelism between the host and SIMD array in any SIMD system. High communication latency has proven to be the largest issue in the implementation of 3D rendering algorithms on this hardware platform. This chapter discusses the issues and provides possible solutions for reducing overhead in future hardware revisions.

Chapter 7
Conclusions

Two implementations of 3D rendering algorithms on a low power SIMD architecture have been presented. The ideal target would be real time 3D rendering applications suitable for a mobile low power system. The Atsana J2210 architecture was used as the development platform for its embedded SIMD Array Processor, and host ARM 922T. These two processors are comparable for a low power embedded application and used to compare results on the rendering algorithms. Results indicate that while the Array Processor inside the J2210 is quite capable of rendering performance and power improvements, communication issues cause the J2210 as a whole to be inappropriate for real time rendering.

Chapter 4 describes implementation of the tile based rendering solution and provides speed and energy performance measurements taken from hardware. This solution effectively uses the Array Processor as an intelligent frame buffer. While the implementation is successful the measured results vary widely in terms of computation time and power consumption. While the peak performance is

10.5× faster than the ARM 922T consuming 8.4× less power, more realistic cases show that work load imbalance can greatly reduce the efficiency of the algorithm.

Chapter 5 describes implementation of the Array Processor as a low level processor for pixel rendering. This implementation is much more successful, showing that on small patches the Array Processor can outperform the ARM in speed, and use up to 7.7× less energy and 5× the performance. However, it is important to note that this implementation has severe problems when dealing with communication bandwidth. The overhead of using the Array Processor as a parallel ALU outweighs the actual performance benefit by a significant factor, since it requires 23 writes per pixel at more than 100 cycles per write for a ~1000 cycle operation. While the Array Processor itself is quite effective, the J2210 platform as a whole is incapable producing improved energy and speed improvements on this algorithm.

Chapter 6 discusses possible communication improvements in order to rectify the problems found in both rendering implementations. System hardware modifications are discussed for future revisions. DMA hardware in order to alleviate the communication issues is suggested to be the best solution.

7.1 Future Research Directions

Both of the presented algorithms were examined in relatively controlled test cases. It would be beneficial to hook up the algorithms to real scenes from real

time rendering software in order to gain more accurate results. In the chapter 4 tile based algorithm CU utilization benchmarks on real applications could provide information on CU workload distribution and speedups. In the Chapter 5 pixel rendering algorithm it would be useful to see both the effect of varying patch size on the speed and energy consumption.

It may be useful to provide a divided approach between that ARM and AP in the tile based method, using the AP for large triangles and the ARM for smaller triangles. There can be a threshold which determines if the AP is being actively used enough, or most of its cycles are being wasted, and remaining triangles can be processed by the ARM. Even a combination of tile and pixel based rendering, both on the array processor, could ease bottlenecks and issues with the algorithms.

The most important concerns found during this research are data communication related. Both algorithms were hurt by the lack of a DMA and slow communication between the SIMD and ARM processors. There is significant future research available in devising a system for transferring rendering data between the Array Processor and host processor in a 3D rendering system. Automated hardware could significantly reduce the load on the system and host processor. With reduced latency when accessing the Array Processor memory, it may be possible for the ARM to store all of its rendering data inside the Array Processor, so that data movement would not be necessary at all. Additional DMA hardware between the Array Processor and display output

hardware would greatly benefit the tile based algorithm, eliminating the need to copy all the data out into system memory prior to displaying.

The Array Processor itself can be modified for better rendering suitability. Studying the best ratio of processor to memory inside the Array Processor may provide useful insight into the efficiency of these algorithms. This is not a simple issue to tackle however, as more processing elements require more data movement from the host.

An alternative approach would be to design an Array Processor with much less embedded memory, only a handful of registers. Reducing the size of the memory array may decrease the latency for host to AP communications. This implementation may be extremely suitable to the pixel based approach described in chapter 5, since it does not use much storage. In addition it may be worth investigating the trade-off of silicon area to a more advanced ALU hardware for multiplication, which is the most time consuming operation in the pixel algorithm. Modifying the CUs themselves for larger bit width (16 or 32) would provide more precision and reduce the amount of data reshuffling required when communicating with the ARM host.

Bibliography

[1] Robert J. Baron , Lee Higbie, "Computer Architecture; Case Studies", Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1992

[2] Intel® Pentium Processor with MMX™ Technology Documentation. Intel Corporation, Santa Clara, CA. [Online]. Available: http://www.intel.com/design/archives/Processors/mmx/

[3] Intel® Streaming SIMD Extensions 4 (SSE4) Instruction Set Innovation. Intel Corporation, Santa Clara, CA. [Online]. Available: http://www.intel.com/technology/architecture-silicon/sse4-instructions/index.htm

[4] H. Fuchs, et al. "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor Enhanced Memories," ACM Computer Graphics, Volume 32, Number 3, July 1989

[5] Deering, M. et al. "FBRAM: A new Form of Memory Optimized for 3D Graphics," Proceedings of the 21st annual conference on Computer graphics and interactive techniques, July 1994

[6] WIKIPEDIA, free online encyclopaedia "Geforce 8 Series Technical Summary". [Online]. Available: http://en.wikipedia.org/wiki/GeForce_8#Technical_Summary

[7] Gene Amdahl, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities", AFIPS Conference Proceedings, (30), pp. 483-485, 1967.

[8] D. G. Elliott, M. Stumm, W. M. Snelgrove, C. Cojocaru, and R. Mckenzie, "Computational RAM: Implementing Processors in Memory," *IEEE Des. Test. Comput.*, vol. 16, no. 1, pp. 32–41, Jan. 1999.

[9] D. Elliott, "Computational RAM: A Memory-SIMD Hybrid," Ph.D. dissertation, University of Toronto, 1998

[10] S. Savchenko, "3D Graphics Programming: Games and Beyond", Sams Publishing, July 2000

[11] Wylie, C, Romney, G W, Evans, D C, and Erdahl, A, "Halftone Perspective Drawings by Computer," Proc. AFIPS FJCC 1967, Vol. 31, 49

[12] Jack E. Bresenham, "Algorithm for computer control of a digital plotter", *IBM Systems Journal*, Vol. 4, No.1, January 1965, pp. 25-30

[13] T. Akenine-Möller, E. Haines, "Real-Time Rendering", 2nd Edition, A K Peters, Ltd., 2002

[14] E. Lengyel, "Mathematics for 3D Game Programming & Computer Graphics," Charles River Media, 2002

[15] Phong, B.T. "Illumination for Computer-Generated Images", Ph.D. dissertation, Department of Computer Science, University of Utah, Salt Lake City. July 1973

[16] Gouraud, H. "Continuous Shading of Curved Surfaces", IEEE Transactions on Computers, vol. 20, no. 6, pp. 623-628. June 1971

[17] J2210 Software Tools User Manual, Atsana Semiconductor Corporation, Ottawa, Ontario, July 2003, Document number: SWE-001-04.

[18] Steve Baker, "The History of The Teapot" [Online]. Available: http://www.sjbaker.org/wiki/index.php?title=The_History_of_The_Teapot

[19] M. Castellón, "A Low Power Parallel Processor Implementation Of A Turbo Decoder," Master's thesis, University of Alberta, Edmonton, Alberta, Spring 2006

[20] K. Breen, J. Tapia, D. Elliott, "Implementation of Three SIMD Algorithms for Graphical User Interface Processing in Mobile Devices Using the Atsana J2210 Media Processor," IEEE CCECE, Saskatoon May 2005