Real Time Depth Map Estimation on Cell

by

Stephen Cavanagh,

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science

University of Dublin, Trinity College

September 2009

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Stephen Cavanagh

September 9, 2009

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Stephen Cavanagh

September 9, 2009

Acknowledgments

Thanks to my father for encouraging me and supporting me through my meandering education

Thanks to my sister for taking the time off work to proof read my thesis

Thanks to Lauren for her endless support during these long days, i will figure out the comma some day

Thanks to my supervisor Michael Manzke for introducing me to the ever challenging topic of programming on the cell

Stephen Cavanagh

University of Dublin, Trinity College
September 2009

Real Time Depth Map Estimation on Cell

Publication No. _____

Stephen Cavanagh
University of Dublin, Trinity College, 2009

Supervisor: Michael Manzke

The aim of this project is to take advantage of the cell processor to process high resolution stereo images with real time, and near real time, frame rates. This thesis will describe the development of a stereo algorithm designed to exploit the power of the cell processor. The stereo algorithm produces a depth map which can be used in many areas including: robotics, movie post production, medical imaging, and augmented reality.

The field of stereo computer vision has become an active area of research over the past number of years. With the advent of new parallel technologies such as general purpose GPU and the cell processor, new avenues and possibilities have opened for real time applications of computer vision.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction to the Topic

The field of stereo computer vision has become an active area of research over the past number of years. This is primarily due to the development and availability of high speed and affordable processors. In the past, the lack of sufficient processing power was a major limiting factor in image processing, and quality real time processing was simply not possible. With the advent of new parallel technologies such as general purpose GPU and the cell processor, new avenues and possibilities have opened for real time applications of computer vision.

The cell processor is a multi-core processor developed jointly by Sony, Toshiba, and IBM. The processor is made up of a single Power Processor Element (PPE) and eight Synergistic Processor Elements (SPE). Each SPE is a RISC processor with 128 bit wide registers and is optimized for Single Instruction Multiple Data (SIMD) processing which is well suited for computer vision and image processing.

The cell processor allows programmers to take advantage of parallel architectures to process computationally expensive operations in significantly less time than using other contemporary processors. Parallel processing is of interest to many fields in computer science, but it is particularly well suited for image processing and data parallel tasks because it is possible to segment the data into small chunks which can be operated on at the same time.

The aim of this project is to take advantage of the cell processor to process high resolution stereo images with real time, and near real time, frame rates.

1.2 Introduction to the Thesis

This thesis will describe the development of a stereo algorithm designed to exploit the power of the cell processor. The stereo algorithm produces a depth map which can be used in many areas including: robotics, movie post production, medical imaging, and augmented reality. The specific goals of this project are to:

- Evaluate the current state of the art in stereo methods and choose an efficient algorithm to implement and test
- Implement a modular framework to efficiently create disparity maps based on local stereo algorithms
- Exploit the power of the cell processor by parallelizing current techniques and optimizing for SIMD processing
- Achieve high quality disparity maps
- Produce an algorithm that is capable of running in real time

In the state of the art chapter of this thesis, the current state of the art techniques for stereo vision will be discussed and assessed. Global and local stereo algorithms will be compared and analyzed for implementation on a parallel architecture.

The design chapter will focus on the design of the project framework which will implement the most suitable stereo algorithm. The square window and adaptive weight algorithms are introduced and examined. Also discussed here are the rationalisations of specific design choices.

In the implementation chapter, the process of developing the project from the design outline to the complete and final implementation will be discussed in detail. Issues and problems that arise during development for the cell platform will also be examined, and solutions developed to overcome these problems will be discussed.

The evaluation chapter will focus on several different aspects of the implementation to provide the results of a thorough analysis of the project. Specifically, the quality of the disparity maps produced will be evaluated, along with the speed of the stereo algorithms implemented. In addition, a comparison of the results achieved by the cell processor and those achieved on a x86 processor will be provided.

Finally, this thesis will discuss the conclusions drawn from the assessment and evaluation of the project, and will also detail plans for future work.

Chapter 2

Background & State of the Art

The following sections will first introduce the topics related to computer vision and stereo algorithms, and then discuss the state of the art in development on the cell processor.

2.1 Introduction to Computer Vision

Computer vision aims to duplicate the effect of human vision by electronically perceiving and understanding an image [2]

This project focuses on the use of stereo vision to recover depth information from an image, or "scene". This process is accomplished by comparing two rectified images of a scene and establishing a pixel correspondence between the images[3]. The end result of this matching process is a disparity map, which describes the distance a certain pixel has moved between the images. A disparity map in itself does not contain explicit depth information, instead it contains relative depths between items in the scene.

2.2 Stereo Vision

The field of stereo vision has been the focus of a considerable amount of research in recent years. A comprehensive study of the area was conducted by Scharstein and

Szeliski [4] in 2002. This research was expanded upon in 2007 by Hirschmuller and Scharstein who provided an evaluation of cost functions for stereo matching[5].

This research evaluated the current algorithms and provided a method for comparing algorithms on a known dataset. This dataset provides a number of stereo images with known ground truth values obtained using a structured light technique[5]. This is significant as it allows a platform for algorithms to be assessed objectively, and provides a universal measure by which we can compare the performance of two separate algorithms. The dataset is freely available and has become one of the standard methods of evaluating stereo algorithms against the current state of the art.

All submitted algorithms are evaluated and ranked on the Middlebury college stereo vision website[6]. The rankings shows that the state of the art in this area is quickly changing and the current top of the field are capable of producing disparity maps with sub pixel accuracy and minimal errors.

Stereo algorithms vary in their implementation however it is observed that most stereo algorithms perform (subsets of) the following four steps [4] which will be further examined in the next sections.

1. Initial Matching Cost Computation
2. Cost Aggregation
3. Disparity Computation
4. Disparity Refinement

Stereo algorithms can be broadly divided into two classes introduced below: local and global algorithms.

2.3 Local Algorithms

Most local methods closely follow the four steps introduced above, but employ a number of different methods to compute each stage. The separation of each stage is advantageous as it allows each stage to be treated as a black box, with simple inputs &

outputs that need to be fulfilled for the algorithm as a whole to function. For example the initial matching cost takes as input the stereo image pair and outputs an initial disparity cost for each pixel at each disparity. The methods used to calculate this cost can be changed and optimised without changes being necessary to other stages.

The following sections will discuss each stage by introducing the concept, and examining current state of the art techniques.

Initial Matching Cost Computation

The initial matching cost is a necessary input to most local and global methods of stereo vision. The purpose of this stage is to calculate a simple estimation of the disparity cost for each pixel in the image at each disparity. This calculation is achieved by comparing the images on a pixel by pixel basis for each disparity.

Although a number of methods exist to calculate the initial cost, they will all contain a large number of pixels with costs that do not truly represent the cost of correlation between the two pixels. The reason for this error has a number of factors.

First, assuming perfect conditions, the matching cost for a pair of corresponding pixels should have a value of zero as the intensities should be identical. In practice an image may have a number of pixels with exactly the same intensity value. This will cause multiple matches at different disparities that have a matching cost of zero and will introduce errors because of the ambiguity.

Second, a number of environmental factors may effect the images and cause differences between the left and right image. These factors include: lighting conditions, shadows, specular effects, and differences in image sensors. This introduces the need for an error threshold.

Finally, pixels in the left image may be occluded in the right image which results in further mismatches. This will result in an incorrect pixel being used to calculate the cost and will introduce errors.

A number of initial costing algorithms have been proposed which mostly calculate the difference in intensity values of the pixels. These methods can be applied to both greyscale and colour images, with both returning a single scalar value representing the matching cost for the pair of test pixels.

Scharstein and Szeliski [4] state in their evaluation that the most common dissimilarity measures are Squared intensity Difference (SD) and Absolute intensity Difference (AD) which offer little difference in performance. Different methods have been proposed which are more robust to change in intensity within an image pair, but this is at a cost of reduced discriminating power.

A number of variations exist for these basic matching costs including Truncated and Scaled Absolute intensity Difference (TSAD) [7]. TSAD differs from the AD method in that values above a set level are truncated, and the result is scaled to use the full range of the chosen storage datatype. The truncated and scaled AD method is the one chosen for implementation in this paper due to its efficient use of memory while maintaining accuracy.

Other matching costs exist including normalized cross-correlation and binary matching costs which return a binary match/no match result that can be useful in some cases. For a complete overview of the advantages and disadvantages of most matching costs please see [4].

Cost Aggregation

In order to reduce errors and increase smoothness, most local methods perform a cost aggregation step which refines the cost of each disparity pixel. The cost aggregation step has received a significant amount of research and the method implemented will have a large effect both on the quality of the end result and also on the time required for computation. Many methods of cost aggregation exist, but this project will focus on aggregation algorithms that utilize a support window around the target pixel to refine the cost.

Window based aggregation algorithms differ in the way that the window is chosen and also how they calculate the cost based on this window. Basic algorithms refine the initial cost values by summing all the values within a square window around the pixel being inspected while more advanced methods have adaptive windows of different sizes and shapes.

Gong et al [8] provide a detailed evaluation of window based aggregation algorithms which are suitable for real time application. We will briefly examine a number of these

methods and discuss their advantages and disadvantages.

- **Square Window** The Square Window approach is the simplest algorithm based on this technique. The aggregated cost is calculated by summing the matching cost of each pixel within a square box surrounding the target pixel. This value may be stored as a whole, or normalized to reduce memory usage. The main advantage of this method is that it is simple to implement and is among the cheapest to compute.

  The main disadvantage of this algorithm is that it assumes that each pixel within the window has the same disparity as the centre pixel[8]. This assumption results in errors in the final disparity image at depth discontinuities. These errors begin to increase as the size of the window increases which puts a limit on the effectiveness of this approach.

  The size of the window chosen will have a direct effect on the quality of the results. A window that is too small will not adequately distinguish the target pixel from other similar results, while too large a window will produce large errors around the borders of objects.

- **Shiftable Window** The shiftable window approach attempts to overcome the errors produced at depth discontinuities by positioning the target pixel (the pixel which is being refined) in different positions within the square rather than simply placing it at the centre. Each position is evaluated and the lowest cost is chosen as the new value for the pixel. This method improves upon the fixed square window approach, but it will still introduce errors with larger windows that crosses depth discontinuities. This puts a limit on the effectiveness of the approach because increasing the window size is important to increase the signal to noise ratio.

- **Adaptive Window** The adaptive window approach attempts to address the issue of window size. The algorithm adjusts the size of the support window to include enough intensity values to adequately distinguish the region, and also selects a window that is small enough to reduce the effect of crossing depth discontinuities.

  The adaptive window approach was introduced as early as 1991 by Kanade and

Okutomi [9] and continues to be an area of active research. Kanade and Okutomi iteratively change the size and shape of the window based on local variation of intensity and disparity estimates. Gong and Yang [10] use an edge detection algorithm, and Wang et al [11] use a radical calculation motivated by experiences of human vision. These algorithms differ from the previous window based approaches because they examine both the intensity values and disparity depths within the region in order to select an appropriate window size.

This extra information allows the appropriate window size to be chosen more intelligently and results in a disparity map with sharper edges. As larger window sizes are selected for areas without depth discontinuities, it is also possible to produce better matches in regions with low texturing or distinguishing features.

The main disadvantage to the adaptive window approach is the additional computational cost required to calculate the size of the window. Selecting an appropriate algorithm to determine the correct window size is also important and the settings used are often heavily dependent on the image being examined. Also a number of methods require a pre processing step to prepare the image which may not be suitable for real time applications[8].

- **Adaptive Weight** The adaptive weight approach proposed by Yoon and Kweon [7] attempts to address all the issues above with a different approach to the adaptive window method. Instead of adjusting the size and shape of the support window as is done in the previous methods, the adaptive weight algorithm uses a fixed window size and instead applies weighting values to the pixels within the windows to determine how much each pixel will contribute to the final sum. The algorithm "computes the support-weights of the pixels in a given support window using colour similarity and geometric proximity" [7].

  This method has a marked advantage over the previous methods as it does not use the initial disparity cost in the calculation of the support window weights. This is an advantage because the initial disparity cost is inherently low quality with many incorrect pixels. This low quality means that the initial cost should be avoided as a basis for calculating the support window.

  By calculating the support window based on colour similarity the method is

similar to many global methods which segment the image based on similar information. However, it is possible to compute this locally and hence is suitable for parallel implementation. These factors produce an algorithm that maintains sharp edges and is capable of computing disparity maps which rival the quality of many global methods. The tradeoff for this increased quality is the increased computational cost required to create and apply the windows with associated weights for each pixel.

Disparity computation

The disparity computation stage is usually a relatively simple process for local methods. The end result of the initial cost stage is a 3D volume, known as a disparity space volume, containing the costs for each pixel correlation at each disparity. The stereo aggregation stage will have refined the costs stored within this volume to reduce the effect of noise. The task of the disparity computation stage is to process this data and search for the lowest cost within the volume for each pixel in the reference image. The disparity computation algorithm is implemented using a local Winner Takes All (WTA) approach. This simply returns the disparity level with a minimum cost for each pixel.

The output for the disparity computation stage is a 2D disparity map which has been generated from the 3D disparity space. The disparity map is created by assigning each pixel in the image the value of the disparity level which contains the lowest cost.

Disparity refinement

Disparity refinement is a common step in local stereo methods because local methods do not enforce any smoothness constraints on the calculated depth, but instead the disparity map is calculated on a pixel by pixel basis. This per pixel approach allows errors which arise from incorrectly matched pixels to be selected as the correct solution during the WTA computation. These errors appear in the final depth map as single, or small patches, of incorrect depths.

In order to remove these spurious pixels, it is possible to perform a post processing step which uses a filter. This filter is commonly implemented as a mean filter which is

efficient at removing small patches of bad pixels while maintaining sharp edges.

Conclusion

The central problem of local methods is to determine the size, shape, and weight of the support window [11]. Many local methods can be calculated quickly, and while most lack the accuracy of many global methods, recent implementations such as the adaptive weight algorithm can produce results which rival global methods.

The aggregation methods described above provide an introduction to the topic of stereo aggregation and introduce a number of problems and solutions that exist within the topic. What can be observed is that the more accurate methods require significantly more processing power. Therefore the use of these techniques in real time applications is a balance between the quality required and performance possible on the target hardware. This requirement varies between applications, for example a slow moving robot may require only a few frames per second of high quality depth information to navigate, while a 3D tracking system may require a high frame rate with a lower quality depth map.

Local methods can be scaled to suit the needs of the application by selecting an appropriate support window size. Furthermore, local methods are also suitable for parallel execution as the cost of a pixel does not rely on segmentation information from the rest of the image. This allows for predictable partitioning of the work load into parallelizable blocks.

2.4 Global Methods

According to the Middlebury rankings, global methods dominate the highest ranked algorithms in accuracy. This is due to the ability of global methods to produce sub pixel accuracy and handle areas which are weakly textured or occluded far better than most local matching algorithms. The disadvantage of most global methods is that they are generally expensive to compute and not suitable for real time applications. However, as processing speeds increase, it is becoming possible to implement some global methods utilizing parallel architectures at close to real time speeds. Yang et al

[12] present a paper which uses the parallel architecture of a GPU to implement a "real time" algorithm which can run at 16fps for a 320 X 240 input image with 16 disparity levels. This shows the algorithms are extremely compute intensive and only usable in real time with low resolution images.

Global methods approach the stereo problem from a different perspective than the local methods, "they make explicit smoothness assumptions and then solve an optimization problem"[4]. This is accomplished by segmenting the input images into regions and matching those regions between the two stereo images. Each region is assigned a plane and the optimization step attempts to fit this plane with the minimum global cost. Global methods tend to skip the cost aggregation step as the majority of the work is accomplished in the initial cost computation stage where energy minimization techniques such as belief propagation, dynamic programming and scanline optimization are applied[13].

Segment based methods are the most successfully applied global method. "These are based on the assumption that the scene structure can be approximated by a set of non-overlapping planes in the disparity space and that each plane is coincident with at least one homogeneous colour segment in the reference image"[13].

The general steps involved in a global stereo method involve:

Image Segmentation

Image segmentation can involve a colour segmentation or other method, such as a simple edge detection segmentation. The goal of the segmentation is to divide the image into regions which will be represented by a plane in the final disparity image. It is assumed for most algorithms that disparity values vary smoothly within a region and that depth discontinuities only occur at region borders[13]. The top performing algorithms at the time of writing segment the image into regions of homogeneous colour, of these algorithms, 4 out of the top 5, [13, 14, 15, 16] use the mean shift algorithm proposed by Comaniciu and Meer [17]. This algorithm has the advantage of using both colour and edge information to determine the regions, unfortunately it is not well suited for parallel execution[15].

Initial Local Cost Computation

A local cost function is applied to create a correlation volume in which a cost is applied to each disparity for each pixel. This step is very similar to the initial cost obtained during a local method and can be accomplished using the algorithms described in that section. This step is approached differently by a number of algorithms. Klaus et al [13] create a disparity map only for reliable points(those which are within a region and not occluded). Wang and Zheng [14] compute the disparity map for the entire image and use a voting system to remove outliers in the final disparity map. Yang et al [15] process the entire image, however, they assign each pixel a value of stable, unstable, or occluded which is then used in the final step to assign priority to the values in each pixel.

Disparity Region Plane Fitting

This stage of the global method is handled differently by most methods, but generally involves fitting a plane to each region obtained during the segmentation stage. There will inevitably be some false matches during the initial cost stage, and in order to deal with these false matches a best fit approach must be taken when creating the plane. Yang et al first fit a plane to the pixels marked as stable, and then propagate this information to the unstable and occluded pixels[15]. In order to reduce the effect of outliers the RANSAC [18] algorithm is used, however, this relies on picking an initial point which is generally chosen at random. If the chosen start point is itself an outlier the results will be adversely affected. Wang and Zheng address this issue in their implementation by using a voting system which computes every possible line created by the points within the region, and obtains the optimal value. This method produces a result which is consistent and does not rely on an initial starting point.

Disparity Refinement

The final step in most global methods involves a refinement stage for the plane which has been constructed. "The purpose of this stage is to increase the accuracy of the disparity plane set by repeating the plane fitting for grouped regions that are dedicated to

the same disparity" [13]. A number of approaches to this refinement are used in global methods, including graph cut techniques such as belief propagation and cooperative optimization

Conclusion

Of the main methods described above none of the high accuracy methods are computed in real time. The function which takes the longest to compute in most cases is the image segmentation task. The segmentation task accounts for 40% of the computation time in [14], and while the exact time is not given by Klaus et al [13], they note that it is the most time consuming step. This large amount of processing time combined with the fact that this algorithm is not well suited to parallel execution makes it a bad candidate for real time applications utilizing the power of a GPU or the cell processor

2.5 Cell Broadband Engine

The Cell Broadband Engine is a multi-core processor developed jointly by Sony, Toshiba and IBM. The processor is made up of a single Power Processor Element (PPE) and eight Synergistic Processor Elements (SPE). The PPE is based on the PowerPc processor and is capable of running code compiled for the PowerPc. Each SPE is a RISC processor with 128 bit wide registers and is optimized for Single Instruction Multiple Data (SIMD) processing. The SPEs are connected to the PPE by an internal high speed bus called the Element Interconnect Bus(EIB).

The SPEs do not contain a traditional cached memory structure as is common in most CPUs, instead each SPE contains a "Local Store" used to store both instructions and data. The local store for each SPE is only 256KB large, so care must be taken to conform to this strict memory limit when designing tasks to run on the SPE. The SPE is able to have data transferred to the local store either by a PPE initialized DMA operation or a SPE initialized DMA operation. For the latter the 64bit memory address, known as the Effective Address (EA) of the data required must be passed to the SPE by the PPE before it can request the data.

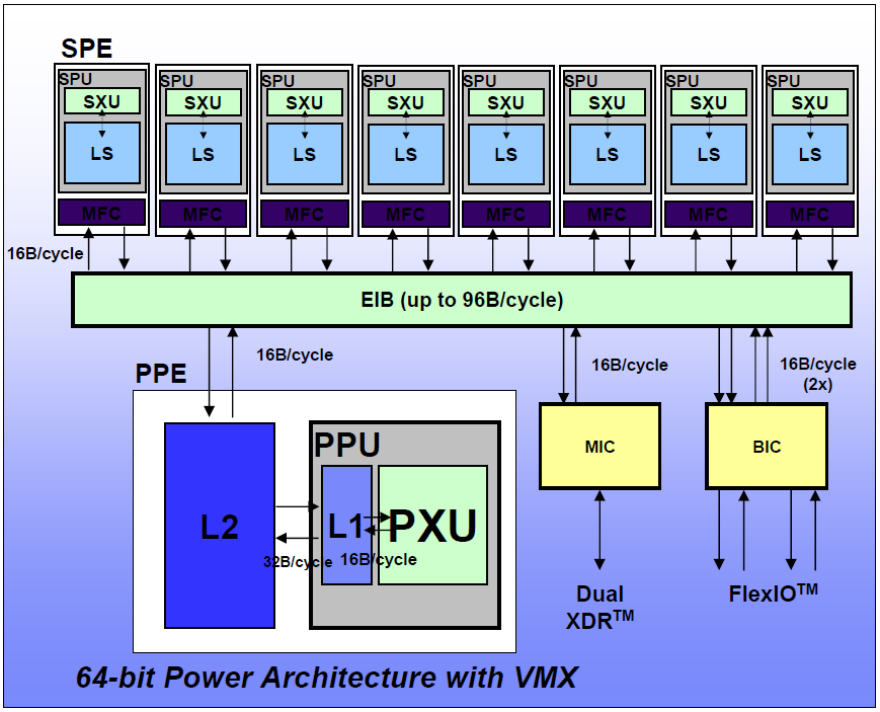


Figure 2.1: Diagram of the cell processor

The PPE is based on a traditional PowerPc architecture and as such is capable of running the linux operating system which provides a high level interface for programming on the cell processor. In order to take advantage of the power of the SPE processing units, a special library is available to run jobs on the SPE. The PPU acts as a controller for each of the SPEs allocating workloads and data.

The cell processor is available in a number of formats, most commonly the ps3 contains a cell processor, however, this only makes 6 SPEs available to the programmer. Alternatively the cell processor is also found in IBM BladeCenter where all eight SPEs are available and many BladeCenters are configured with 2 cell processors in a single machine allowing the use of up to 16 SPEs in a single application.

The SPEs provide an excellent platform for developing computer vision related applications because of the extremely wide 128 bit registers and SIMD capabilities. For example a single SPE can perform a calculation on 16 - 8 byte integer values in a single operation. If all SPEs are taken into account that gives 128 - 8 byte calculations performed per tick of the processor.

The only limitations of this vector processing is that it performs a single instruction on all 16 values. For example, adding a list of characters to another list of characters. This may not always be possible for many types of computation, but in the computer vision field it is often required to perform the same operation hundreds or thousands of times on similar data.

2.6 Cell Applications

OpenCV on Cell

The OpenCV library is an open source project which is one of the most widely used and extensive image processing libraries available. The OpenCV on Cell project is an implementation of this library optimized for the cell processor. As shown by Sugano and Miyamoto in their paper comparing the performance of the cell processor to an Intel Core 2 Duo E6850, the use of the cell processer has proved to significantly increase the performance of most functions provided by the library[19].

This paper shows that the cell implementations of these functions are as much as 14 times faster than the original code. Sugano and Miyamoto noted that the cell implementations are heavily optimized while some of the original OpenCV functions may not have such a high level of optimization. The OpenCV library also includes the Intel Performance Primitive (IPP) which is a subset of the openCV functionality provided in a closed source optimized implementation. Sugano and Miyamoto show that even these optimized functions are out performed by the cell processor, for example the cvPyrDown function is 4.8 times faster on the cell implementation. Figure 2.2 shows the speedup gained by a number of functions implemented on the cell processor.

OpenCV provides many functions related to this project, however, the most relevant is cvFindStereoCorrespondence. This function takes as input a pair of rectified images and outputs the generated disparity map. The stereo function implemented is based on an algorithm presented by Birchfield and Tomasi [20]. This algorithm examines the images in a local pixel by pixel method, alocating a cost to the difference between pixels. The simplicity of the algorithm combined with the power of the cell processor makes it suitable for real time application with computation time for a 1390x1110 image

Target functions	Plan		Status	Performance (optimized / original) 0 100 % fast slow <----- ----->	Comment
	Schedule	Target Optimization level			
cvAbsDiff	Oct 31, 2007	SIMDize	SIMDized	4.1 %	
cvAbsDiffS	Nov 30, 2007	SIMDize	SIMDized	3.1 %	
cvAdd	Oct 31, 2007	SIMDize	SIMDized	3.9 %	
cvAddS	Nov 30, 2007	SIMDize	SIMDized	3.3 %	
cvAddWeighted	Oct 31, 2007	SIMDize	SIMDized	2.8 %	
cvAnd	Oct 31, 2007	SIMDize	SIMDized	8.6 %	

Figure 2.2: OpenCV Cell performance overview[1]

being only 84ms. While the method presented by Birchfield and Tomasi is suitable for real time applications, like many local methods, it suffers from a lack of accuracy near depth discontinuities. Since its publication in 1998, a large amount of research has been completed in this area and this paper aims to produce a higher quality disparity map than the one provided by the OpenCV library.

Folding@home

The folding@home project is a distributed computing system that was first released in 2000 with the goal of providing computing resources for the simulation of protein folding. The project was ported to exploit the power of the Playstation 3 in 2007 and now receives a large chunk of its processing power from the cell processors volunteered by many PS3 owners.

The project is an excellent example of the power of the cell processor. According to Beberg et al [21] a standard single core PC is capable of simulating only 20 nanoseconds for a small protein per day. This is compared to the cell processor which can simulate as much as 500 nanoseconds per day for small proteins.

The folding@home project consists of approximately 400,000 hosts which actively return work to the servers. Of those hosts, only 50,000 are PS3s, however, the PS3s

contribute approximately 30% of the processing power [21].

The project has also been ported to function on GPUs with Nvidia GPUs now contributing the largest amount of processing power to the project, however, Beberg et al note that the GPU is also the most limited in terms of what can be computed on the platform. The PS3 platform is also somewhat limited, but is more versatile than the GPU.

Cell SDK

The Cell SDK provided by IBM[22] includes a large number of example programs that demonstrate the use of the cell processor including simple examples of PPU only code all the way to complex examples using multiple SPUs, DMA and ALF examples. The SDK should be considered among the primary resources for starting development on the cell processor.

Chapter 3

Design

The focus of this project is to develop an efficient modular framework for the creation of dense disparity maps from stereo images on the Cell BE platform. The project will take a rectified set of images as input and will output a disparity map for the reference image.

In order to take full advantage of the cell processor each stage of the algorithm will be adapted to a parallel architecture and optimized for SIMD processing.

The final project will be evaluated for both speed and the quality of disparity map produced against the Middlebury stereo dataset[6].

The following sections will first describe the goals and basic plan for the project and then explain the core concepts and design decisions chosen for this project.

3.1 Goals

- To evaluate the current state of the art in stereo methods and choose an efficient algorithm to implement and test
- To implement a modular framework to efficiently create disparity maps based on local stereo algorithms
- To exploit the power of the cell processor by parallelizing current techniques and optimizing for SIMD processing

- To achieve high quality disparity maps
- To produce an algorithm that is capable of running in real time

3.2 Stereo Algorithm

The initial sections of this paper evaluated a number of the state of the art methods for stereo correspondence including both local and global methods. In order to create an application that will best fit the project goals a selection must be made from these algorithms for implementation on the cell processor.

Two main factors indicate global algorithms are not suitable to achieve the goals of this project. First, there is a high computational cost associated with global methods that will negatively impact the applications ability to run in real time. Second, a necessity for many global methods is the mean shift algorithm, which is fundamentally unsuitable for implementation on a parallel architecture.

This leaves the local algorithms to choose from due to their high speed and ability to be calculated in parallel. Local methods also have the advantage that most implementations follow the same four basic steps which fits well with the projects goal of a modular design. This allows two variations of the project to be built with only a small section of separate code. Therefore, both a standard square window approach and also the adaptive weight algorithm proposed by [7] will be developed.

The square window implementation will demonstrate a real time but lower quality method, while the adaptive weight implementation will demonstrate the higher quality disparity map with the cost of slower computation.

A number of modifications to the original adaptive weight algorithm are required to suit the parallel architecture and memory usage on the SPUs which will be described in the implementation section.

3.3 Project Plan

The primary goal of this project is to build a framework which exploits the processing power of the cell architecture in creating a disparity map. In order for a program to use the full power of the cell processor it is necessary to segment the tasks used for creating the disparity map into blocks which can be implemented in parallel.

While it is possible to directly implement the algorithm for a parallel architecture, the first stage of this project involves creating a serial implementation of the entire process. This forms both a basis for the segmentation into a parallel architecture and a means of testing the performance gains achieved on the cell processor.

The initial implementation will be written in the C programming language, but will not use any code specific to the cell platform. This code will be capable of running on both the powerpc based PPU on the cell processor, and also on the x86 architecture allowing further comparison of speed between the different platforms.

The initial version will include the creation of a pipeline of functions which will be created as separate modules that provide a black box approach to the design. In other words, each function will expose a set of inputs and outputs while keeping the actual implementation of the algorithm completely independent of all other functions.

The second stage of the process will involve creating a cell platform specific implementation. Each module will be segmented into blocks and designed to run on multiple SPUs within the cell processor. The modular design provides an advantage as it is possible to convert and optimise each module individually without compromising the other stages in the pipeline.

The final stage of the process is to optimize the parallel tasks to take full advantage of the SIMD instruction set. This process involves arranging data in a suitable way for SIMD instructions and restructuring the code to implement those instructions efficiently.

3.4 Parallel Implementation

The process of exporting functionality to the SPUs will be accomplished using the Accelerated Library Framework (ALF) instead of directly using the low level libspe library.

"The ALF provides a programming environment for data and task parallel libraries and applications"[23]. This framework provides a number of advantages over the traditional libspe library including dynamic load balancing for data parallel tasks, and advanced buffering techniques to efficiently transfer data between main memory and the SPUs local store.

The end result of the project will be a program that is suitable for running on the cell processor and which scales well to efficiently use all the parallel resources provided by the platform, ranging from the six SPUs available on the PS3 to the 16 or more available on the IBM BladeCenter and PCI Xpress addin boards.

3.5 Data Management

A key requirement for development on the cell processor is the appropriate storage of data and the correct transfer of data between main memory and the SPUs local store. Another concern is the segmentation of the data for parallel execution. This memory segmentation is handled differently for each stage of the pipeline and we will address each transfer in the appropriate section.

Further details of the low level memory alignment is presented in the implementation section.

3.6 Algorithm Pipeline

As described in the previous section, most local stereo algorithms can generally be divided into four separate stages. These stages provide a convenient method for segmenting the work into a modular pipeline. Before moving between each stage in the parallel pipeline implementation, it will be necessary for all computation within the

stage to be complete. This is commonly known as a barrier in parallel computation and is necessary to ensure that the correct and complete data is processed during the next step.

3.7 Initial Cost Computation

The initial cost computation produces the initial low quality disparity cost estimates for each pixel in the image, at each disparity level. The end result of this computation is a 3D matrix of costs, known as the disparity space volume which contains the cost for each pixel at each disparity level.

The disparity space will have a height and width equal to the height and width of the reference image, and a depth equal to the number of disparity levels being calculated.

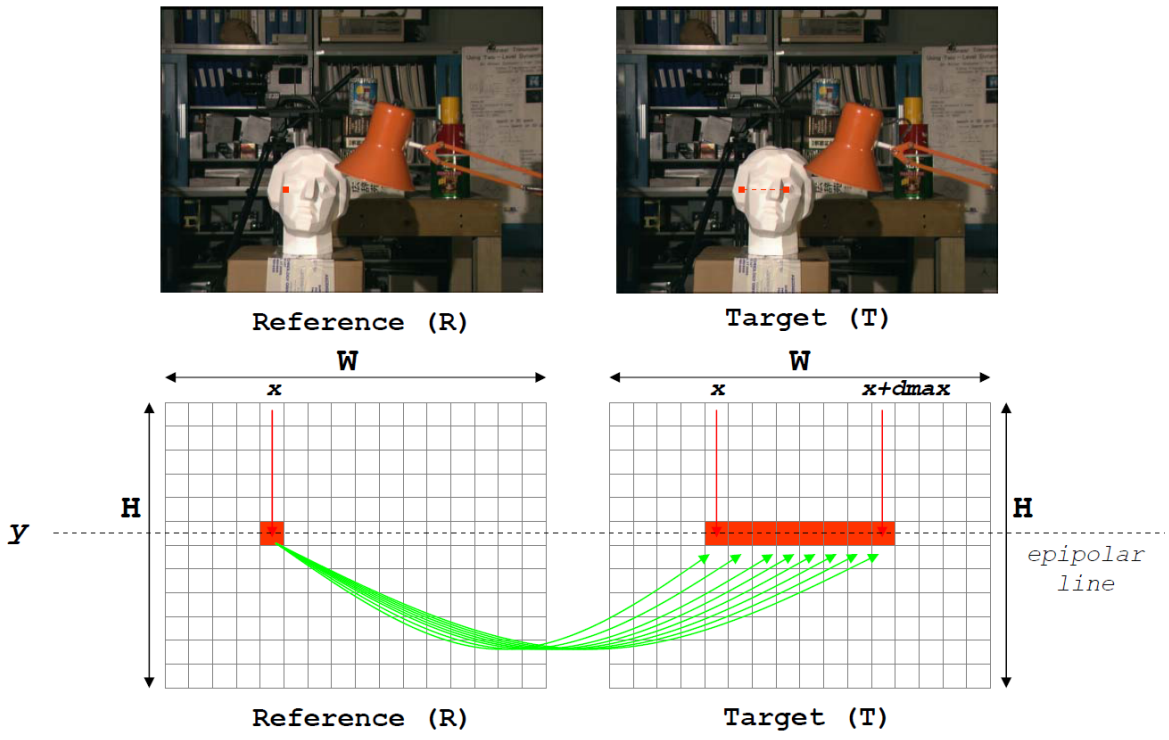


Figure 3.1: Initial Cost Calculation.

Figure 3.1 illustrates the method used to calculate to cost. For each pixel in the reference image the cost is computed by calculating the absolute intensity difference

between that reference pixel and the pixel in the target image offset one pixel for each disparity level.

In order to take maximum advantage of the SIMD processing capabilities of the cell processor, and also to make most effective use of the memory bandwidth, the disparity space will use a single 8 bit integer value to store each cost. This means that the range of costs available is limited to 256 values. To make efficient use of this space and increase the sensitivity of the matching cost, we use the TSAD formula described above to calculate the cost for each corresponding pixel set.

$$C(u,v,d) = min(\left|I^{ref}(u,v) - I^{tar}(u-d,v)\right|, C_{max})\ X\ \frac{255}{C_{max}}$$

Where C(u,v,d) is the cost for a single pixel correspondence, I^{ref} (u, v) is the intensity of the pixel in the reference image, I^{tar} (u - d, v) is the intensity of the pixel in the target image offset by the disparity, and C_{max} is the truncation value.

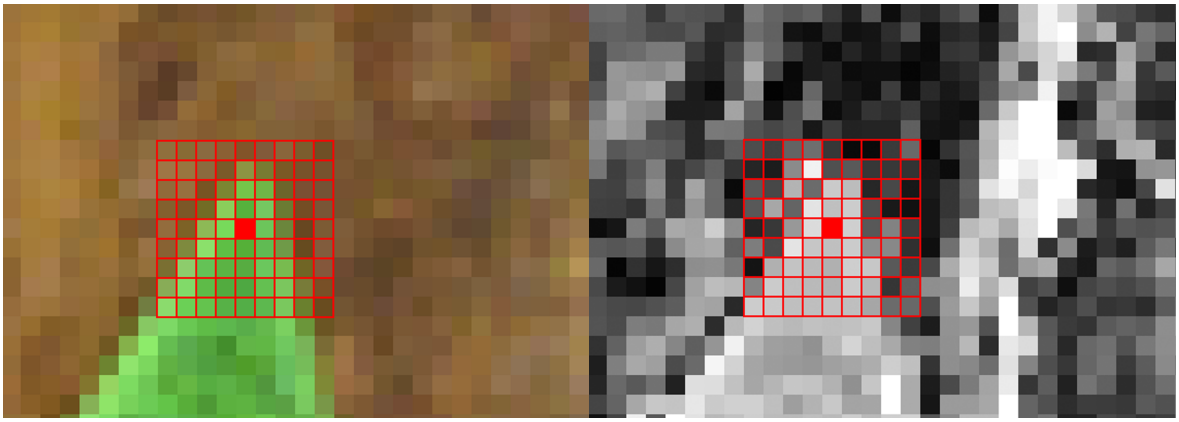
The TSAD algorithm has the advantage that large costs which are unlikely to be the correct disparity are truncated, and this allows more space to store values which may be the lowest cost within the range available. The truncation value must be chosen appropriately in order to avoid truncating values which may be the lowest cost, and also must be set low enough to properly distinguish the correct lowest cost from similar values.

3.8 Cost Aggregation

The cost aggregation stage is the most important step in the stereo algorithm. For this project two separate aggregation methods will be implemented. This will provide a comparison for both quality and speed of the algorithm.

Square Window

The first method is based on the square window approach and is the among the simplest of all local methods. In order to refine the value of a single disparity cost in the 3D disparity space, a support window around the value is defined at a constant disparity



(a) (b)

Figure 3.2: Square Window Aggregation

level. This 2D window can be viewed as a greyscale image which represents the cost of correlation between the reference image and the offset image for a region of pixels at the given disparity level.

Figure 3.2 shows a zoomed in section of a stereo image. This is useful to illustrate the use of a square window approach. The area of the reference image the window encloses can be seen in (a) and (b) shows the associated costs for a single disparity layer. The pixel highlighted in red will have its cost updated with the sum of the cost of all pixels within the window, scaled by the number of pixels within the window.

Adaptive Weight

The second method focuses primarily on quality over the speed of calculation. To that end we will use the adaptive weight algorithm as the basis for the implementation.

The adaptive weight algorithm shares the same basic structure as the square window method described above, the window size is fixed, but rather than simply calculating the sum of each of the costs inside the window, each cost is given a weighting value which determines how appropriate that cost is to the pixel under consideration.

"Visual grouping is very important to form a support window and to compute support-weight and, therefore, the Gestalt principles can be used to compute support-weights"[7]. The two main Gestalt grouping concepts that are used within the adaptive

weight algorithm are similarity and proximity. Based on these Gestalt principles we can write the support weight of a pixel within the window as[7] :

$$w(p,q) = f_s(\Delta c_{pq}).f_p(\Delta g_{pq})$$

where Δc_{pq} and $f_s(\Delta c_{pq})$ represents the intensity difference and strength by similarity repectivly, and Δg_{pq} and $f_p(\Delta g_{pq})$ represents the distance and strength by proximity respectively.

Two modifications to the original algorithm will be made to improve the speed and reduce the memory bandwidth needed for the calculation.

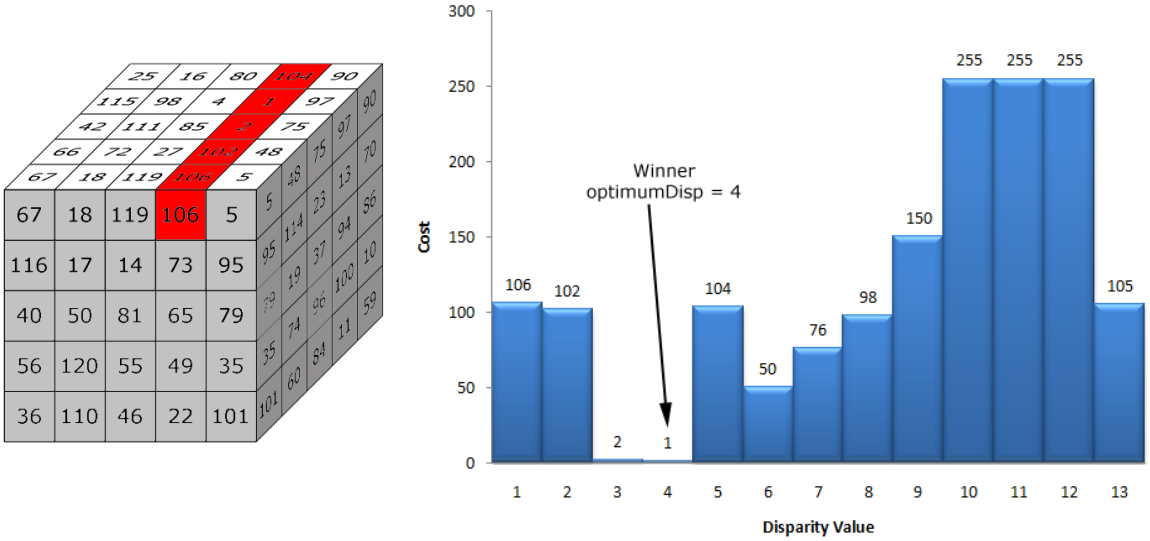
First, the adaptive window weights will be calculated only once based on the reference image. The original algorithm computes new weights for each disparity layer based on the target image at the given disparity. This simplification significantly reduces the computation required for each disparity while still maintaining most of the advantage gained by the adaptive weight algorithm.

Second, the image will only be processed from the point of view of the reference image. This will cause a loss of quality in the regions where occlusion occurs, but will half the number of computations required to produce the final disparity map and also importantly it will reduce the size required for the disparity space volume which will be transfered to and from the SPU memory.

These simplifications provide a significantly quicker implementation while still maintaining the basic principles of the adaptive weight algorithm (weights applied by proximity and similarity). The overall benefits of the adaptive weight algorithm will be maintained in most regions of the image, including the sharp edges and good handling of low texture areas.

3.9 Disparity Computation

The disparity computation step calculates the optimum disparity map using the WTA approach. As described above, the disparity space has the same height and width as the reference image and each element corresponds to a pixel in the reference image.



(a) Section of disparity space

(b) WTA Algorithm operating on a single row

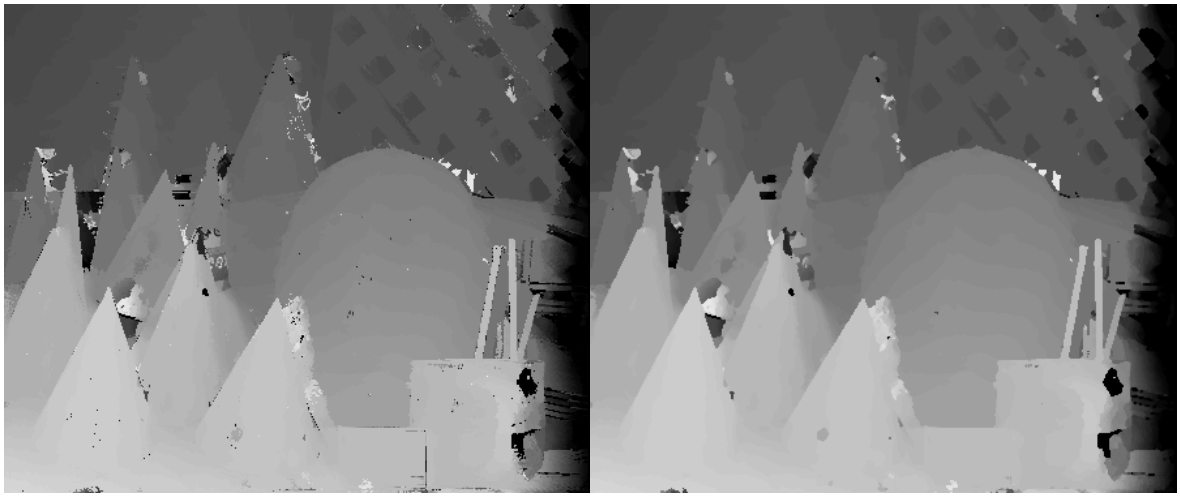
Figure 3.3: Disparity Level Selection

The depth of the disparity space is based on the number of disparity levels that have been evaluated. See figure 3.3 (a) for a cross section of the disparity space matrix, and figure 3.3 (b) for an illustration of the WTA algorithm acting on the data highlighted in (a)

The WTA approach simply examines the disparity space for each pixel coordinate in the reference image and selects the disparity level with the lowest cost. The lowest cost determines the location of the optimum disparity level but its value is not relevant. The optimum disparity map is constructed from the location of the lowest cost for each pixel within the matrix, that is :

$$optimumDisp(u, v) = indexOf(min(d(u, v))$$

Where d(u, v) represents a single line of the disparity space matrix at coordinates (u, v). This output value often scaled to make full use of the data type range and passed to the disparity refinement algorithm.



(a) Unrefined Disparity Cost (b) Disparity after 3X3 Median Filter

Figure 3.4: Disparity cost refining

3.10 Disparity Refinement

As described in the previous chapter local algorithms do not enforce a smoothness constraint on the image and so the final disparity map may contain spurious mismatches. These mismatches appear in the image as a "salt and pepper" noise, with a large variation from the true value. In order to remove these mismatches a median filter will be applied to the output of the disparity computation stage. A median filter is well suited to the job of removing this type of noise because the large difference in the pixel to its surrounding values does not effect the result like it would with a mean filter.

Figure 3.4 shows an example of a disparity map before (a) and after (b). The image on the left can be seen to contain a number of obviously incorrect pixels especially near the border of objects. Image (b) shows the image after the median filter is applied, where most of the small pixel errors have been removed without reducing the sharpness of the borders between regions.

The median filter is applied to each pixel in the disparity map and operates on a 3X3 section around the target pixel. The values of all pixels are arranged and the median value is selected and stored in the target pixel.

3.11 Summary

This section has described the design of the stereo algorithm pipeline that will be developed. The goals for the project have been introduced along with the project plan and algorithms that will be used to achieve those goals.

The issue of designing a parallel implementation has been introduced and finally a description of each stage of the pipeline that will be developed has been provided. This high level overview should allow a complete understanding of the project while the next section will discuss the low level implementation details of the project.

Chapter 4

Implementation

The following section will describe the implementation of the algorithms described in the design section. The target platform will be discussed, along with the framework implementation and specific issues that need to be addressed when developing for the cell processor. The specifics for implementing each stage of the pipeline will also be examined, along with issues that arose during the course of development.

4.1 Introduction

Development on the cell platform is a mix of coding a serial algorithm that will be run on the PPU and segmentation of tasks into parallel units to be executed on the SPUs.

The PPE contains contains a 64 bit, dual threaded PowerPc core. It has a 32KB level 1 cache, 512KB level 2 cache and provides similar functions to traditional CPUs. The PPE is intended primarily for running the operating system, managing system resources and executing tasks on the SPU [24].

The SPE is what does the main chunk of the work in the cell processor. Each cell chip contains eight SPEs which are designed for high speed SIMD operations. Each SPE contains a Memory Flow controller (MFC) and a Synergistic Processor Unit (SPU) and can operate independently of one another.

The SPU does not contain a cache like most traditional processors, instead it contains a 256KB local store (LS) of memory directly on the chip which is used to store

both the program instructions and any data required for the task. The local store must be manually filled with data so that it is ready for the processor when it is required. The MFC enables data transfer between the SPU local store and main memory and vice versa.

SPUs were not designed for general purpose processing, and are not well suited to run operating systems or traditional sequential code. Instead they recieve instructions from the PPU which manages their execution[25].

This implementation follows these general guidelines to structure the work tasks required for developing the final disparity map. The PPE is used only as a managment device and does not do any of the image processing tasks itself. The PPE loads the initial data, divides up tasks into segments that can be processed in parallel and assigns them to the SPUS. The PPE also manages load balancing, task scheduling, and barriers depending on the needs of the current stage being executed.

Once assigned a task by the PPU, the SPU will load the data from main memory, process the chunk of data required, and return the data to main memory.

4.2 Development & Target Platform

This project was targeted at the cell platform, however a PowerPc and X86 version were built during the initial sequential development of the project. This allows for a comparison of results between the different platforms.

Developing applications for the cell processor is currently only supported on the linux platform. IBM provides the cell SDK which contains a number of libraries and examples of applications to develop for the cell platform.

Applications can be compiled either on a X86 processor using the GNU or XLC cross compilers provided by IBM, or directly on the cell hardware. Testing can be carried out by using a remote debugger attached to the machine containing the cell processor.

4.3 Accelerated Library Framework

In order to understand the implementation details, the ALF structure, around which the application is developed must first be described.

The ALF provides a programming environment for data and task parallel libraries and applications. The API provides a set of interfaces to simplify development on multi-core systems[23].

Developing applications for the Cell processor requires the use of custom libraries which can launch object code on SPUs to execute tasks in parallel.

The most common method to accomplish this is to use the libspe library to assign object code to the SPU and run the code as a seperate thread. Libspe allows for messages to be sent by mailboxes and DMA transfers to move data from main memory to the SPU local store.

The advantage of this method is that low level control of the application is gained, however, the programmer is required to handle all load balancing, data transfers, and buffers manually.

The ALF allows the application to implement the same behaviour, however, many of the low level details of SPU handling are managed by the ALF runtime. This allows for easier development of parallel applications with the focus being on the application required rather than on managing the SPUs.

Target Platforms

The ALF is not directly targeted at the Cell platform, hence the use of the name Accelerators and Hosts instead of simply using PPU and SPU. This has the advantage that applications written for the cell processor using the ALF framework are compatable with a range of multiprocessor platforms. The most immediate advantage is the easy integration of the project on the IBM hybrid BladeCenters. The ALF provides an API to query the number of available accelerators and can transparently execute the application taking advantage of all available accelerators.

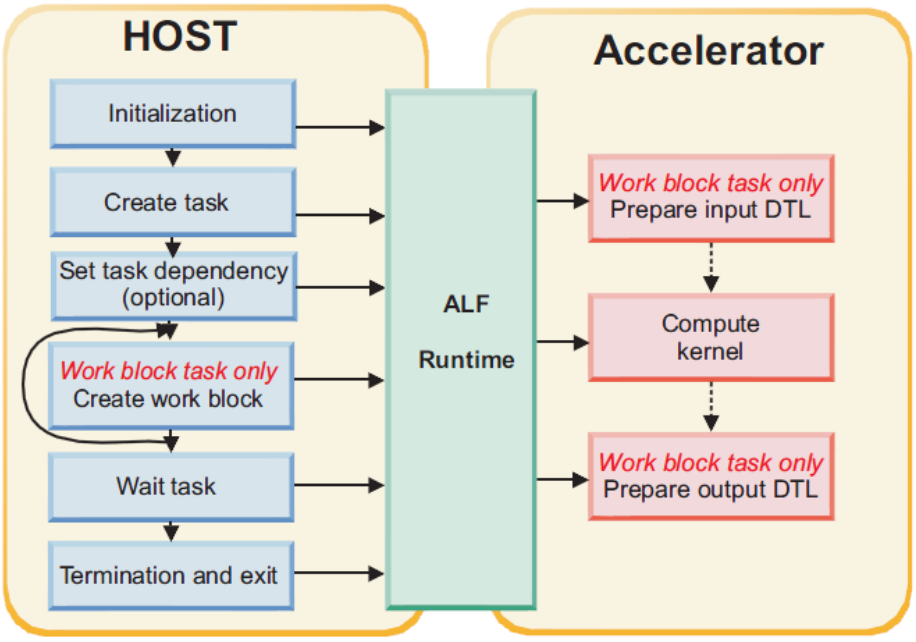


Figure 4.1: ALF Application Structure & Process Flow

ALF Structure

ALF applications must follow a basic structure in order to properly use the ALF runtime to manage the application. There are two main structures defined to describe jobs that are executed by the ALF runtime on the SPUs. Figure 4.1 shows the application structure and process flow of an ALF application.

- A **Task** is an abstract definition of a processing job[25]. It contains specific information about the task to be executed, for example the data that will be transfered, how many SPUs to use, and what compute kernel must be executed. For this application a seperate task is created for each stage of the pipeline described in section 3.6.

  Each task is executed in parallel and workblocks are executed based on a scheduling algorithm provided by the ALF runtime to optimize performance.

  One of the main problems of parallel execution is dealing with data dependencies; it is necessary to ensure that one task has finished with some shared data before the next task can be executed. Each stage of this projects pipeline has such a

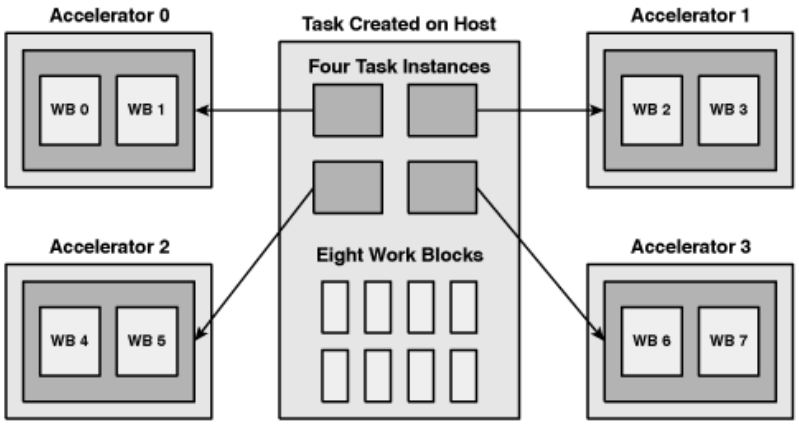


Figure 4.2: ALF Task and Workblocks

requirement, for example the stereo aggregation stage cannot be started until the initial cost is calculated.

In order to execute these tasks in the correct order a barrier is created between each stage. This feature is provided by the ALF API by creating a dependency between tasks. The ALF runtime will then ensure that one task has completely finished before the next is executed.

- A **Workblock** is a concrete invocation of a task that will be executed on the accelerator (SPU). A task may contain multiple work blocks that execute on different data in parallel. A workblock can be compared with a single thread being executed by libspe. Figure 4.2 shows how work blocks exist within tasks and how they are distributed to the accelerators(SPUs).

  Each workblock will have the address of some data that must be processed and a compute kernel which will be applied to that data to produce the output. Workblocks can be either single-use, or multi-use. For multi-use workblocks the workblock remains active on the SPU over multiple iterations executing the same compute kernel but on different data. The compute kernel of a workblock contains the task specific code which must be run on the data in the local store memory.

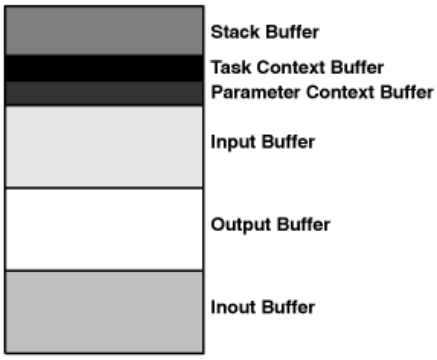


Figure 4.3: ALF Accelerator Memory Management

Memory Managment on SPU

The SPU local store is shared between the application instructions and the application data required for the task. The ALF library manages this memory by dividing up the memory into six main buffers. Figure 4.3 shows the arrangment of the buffers in SPU memory.

The choice of memory buffers has an impact on the performance of the application and the buffering that the ALF runtime can provide, as such it is important to understand their use in order to properly use them in the implementation.

- The **Task Context Buffer** contains common data across multiple workblocks in a single task. The task context is usefull for workblocks that need a common area which can be accessed and updated across workblocks.
- The **Parameter Buffer** is passed to a workblock at the time of its execution and contains specific information about the workblock being executed. For example, the amount of data to be processed and the data's EA in main memory.
- The **Input Buffer** contains the data which has been transfered to the block to be processed. This data will not be transfered back to main memory after the compute kernel finishes.
- The **Output Buffer** is where all data that is to be output from the compute kernel back to main memory must be saved. The output buffer will be transfered by DMA to the target EA in main memory.

- The **Overlapped IO Buffer** is useful when you can overwrite input data with output data to maximize the memory available on the accelerator (SPU). It is tranfered to the SPU before the compute kernel is called and returned to main memory after it executes.
- The **Stack Buffer** contains variables created and used during the computation kernel. It operates similarly to a traditional cache.

Each memory buffer must be configured to a set size by the programmer and to make maximum use of the limited 256KB of memory the values chosen should be exactly the right size for the data required. This is accomplished by explicitly describing the data that each block will need at the initialization of the task. For example, the intial disparity cost task uses workblocks which contain a single row of the image. To accomodate this the input buffer size is calculated using

```
inputBufferSize = imageWidth * channels * sizeof(byte);
outputBufferSize = imageWidth * numLayers * sizeof(disparityLayer)
```

This value is then passed to the task when it is being created and the ALF runtime will allow that much space for the input buffer on the SPU. Assigning the correct value to the buffer is important to avoid over lapping data causing corruption and incorrect results. The buffer size is also important to allow the ALF runtime to select the appropriate DMA buffering solution described in section 4.5.

Data Transfer Lists

The ALF runtime manages the data transfer between main memory and local store of the SPU by creating DMA transfers managed by its built in buffering system. Communication mechanisms like mailboxes and DMA are managed by the ALF runtime, however, the code still has to identify which data needs to be transfered and when.[25].

This is accomplished by creating Data Transfer Lists (DTLs) which are similar to DMA lists in libspe. The DTL tells the ALF runtime where the data exists in main memory, how much to grab, and where to put it in the Local Store. A DTL can contain

a large list of data transfers that will be completed before the computational kernel is executed. Whenever possible the data transfers are completed while another job is being computed on the SPU.

As with libspe, both the PPU and SPU can initiate the data transfer. For applications with a large amount of data transfer it is necessary to use the SPU to pull data from main memory, rather than push it from the PPU side. This has the advantage of freeing up the PPU to manage the tasks more appropriately, and is can be quicker because each SPE can execute 16 DMAs in parallel[26] compared to the PPE which only has a DMA request queue size of 8 to be shared over all SPUs.

Since DTLs use DMA transfers to get the data onto the SPU, they are subject to the same restritions in size and memory alignment as those created using libspe.

ALF Summary

The ALF framework provides a number of useful functions to enable parallel task execution and efficient buffering of data transfers. It is well suited for developing a image processing application which features large amounts of data requiring a single compute kernel to operate on them. The ALFs task dependencies also allow for the efficient implementation of the seperate stages of the stereo algorithm.

The library also provides a robust development environment allowing development time to be spent on the application rather than on low level code. For these reasons the ALF was chosen for this application over simply using the libspe libraries for parallel task execution.

4.4 Application Framework & Pipeline

One of the core goals of this project was to implement a framework for local stereo algorithms. The modular design of the application allows for each component to be changed without any effect on any other components. This is shown with the implementation of the two seperate aggregation techniques which simply require selecting a different computation kernel at runtime.

The framework developed also provides the basic setup and image segmentation techniques required by many other image processing tasks.

The seperation of each stage of the pipeline into seperate modules requires each module to be performed in a seperate ALF task. Each task will need to return extra data to main memory to be used as input to the next stage of the pipeline. This causes an increase in the amount of data tranfer needed to complete the final disparity map, however, by computing each stage in a specific method suited to the task, we are able to cut down on the computation required by seperating each function.

In order to evaluate whether this modular framework design affects the final performance of the application, a single combined SPU task was created that takes as input the blocks of reference images and outputs the final disparity map. This comparison is discussed further in the evaluation section.

Parallel Task Implementation

The process of segmenting the application into blocks which can be processed using the parallel architecture of the cell can be approached in a number of ways. Figure 4.4 shows an example of the two main methods of parallizing the tasks which are explained below.

- A **Multi Stage Pipeline** is created by assigning a specific job in the pipeline to each SPU. The SPU will always carry out the same operations on data that is given to it and then output the data to the next SPU in the pipeline. This has the advantage that you do not need to reload the program data to the SPU for each task, the task itself does not need to be parallelized, and also that we are taking advantage of the SPU to SPU communications instead of reading from main memory.
- **Task Parallel Execution** involves parallelizing each task and implementing that task as a group of workblocks that can be processed in parallel. Only after all tasks are complete is the object code on the SPU replaced with the code for the next task. This has the advantage of easy load balancing because any available SPU can be given data to process, however, it is only possible when the task can be processed in parallel.

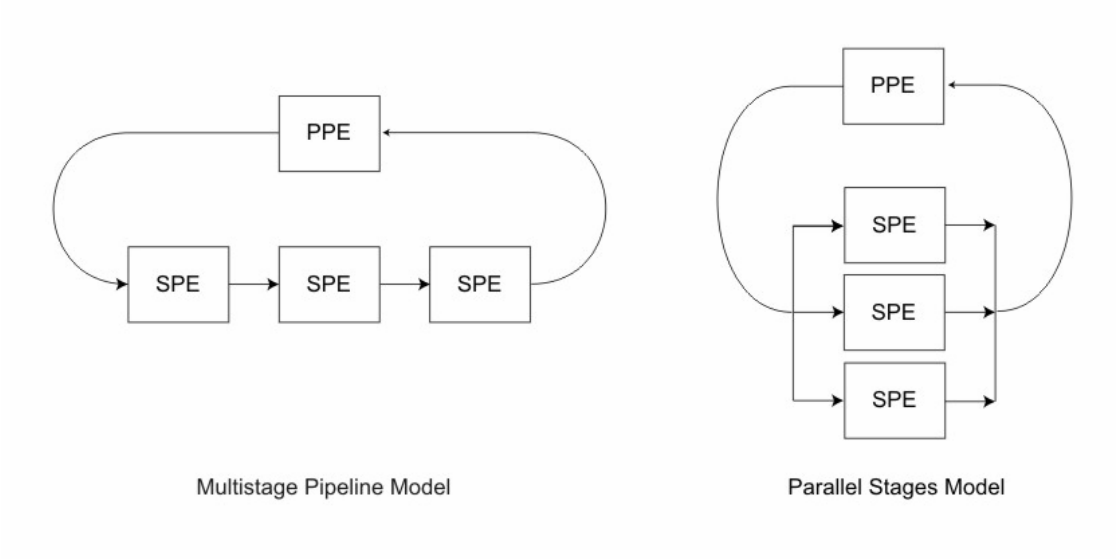


Figure 4.4: Parallel Implementation Models

- **PPE Centric Model** This involves assigning a specific task to each SPU in a similar manner to the multi stage pipeline, however, in the PPE centric model there is no SPU to SPU communication. The PPE simply executes data on a specific SPU as needed, returning its results to main memory.

Multi-stage pipelining is typically avoided because of the difficulty of load balancing. In addition, the multi-stage model increases the data-movement requirement because data must be moved for each stage of the pipeline" [27].

The PPE Centric model relies heavily on the PPU to provide SPUs with jobs and this slows down the execution of the tasks. There is also no segmentation of task so it is difficult to ensure that the data is ready for the task being executed. By statically assigning a specific job to each SPU we are also losing the processing power of the SPUs without the correct task for the job required.

The chosen method depends heavily on the type of application being implemented. This implementation uses a local stereo method to compute the disparity map which means that there are limited dependencies between pixels and it is possible to successfully segment the tasks into parallel work blocks. Due to the ease of segmenting the tasks and the added speed achievable from task parallel execution, this is clearly the best method for this type of application.

The remaining consideration is whether to use a single compute kernel to process from start to finish or to use multiple compute kernels. To fit with the modular framework of the design, multiple compute kernels are developed each with a specific function in the pipeline. ALF task dependencies are used to ensure that each task is finished before proceding to the next stage in the application pipline. In order to evaluate the effect of this modular pipeline, a single stage parallel task is also implemented by combining all stages into a single compute kernel.

4.5 Memory

Communication is a crucial aspect of programming any multi-core processor. Without efficient data transfer between processing elements, the application cannot take full advantage of the device and stalls will be caused while waiting for data to load[25].

For this application we use SPU controlled DMA to pull data from main memory. This is achieved by creating DMA Lists which contain the Effective Address (EA) of data in main memory, the location that the data should be stored in the Local Store (LS) and the amount of data to transfer. A DMA List can contain a number of these entries to gather data spread out in main memory into a continous block in the LS.

The transfer of data to the LS is managed by the Memory Flow Controller (MFC) on each SPE. The MFC is connected to the Element Interconnect Bus (EIB) which connects the SPE to all other SPEs, PPE, and main memory. The EIB provides a high speed bus allowing communication between all the elements of the processor.

The following sections describe a number of areas relating to memory management issues and solutions addressed during implementation.

Data Layout in main memory

The structure of image data in memory has a large role to play in the overall structure of an application targeted at the cell platform. The cell processor is capable of vector processing and taking advantage of this is essential in order to properly exploit the power of the chip. A vector is an instruction operand containing a set of data elements

packed into a one-dimensional array[28]. Both the PPU and the SPU can operate on vectors, however, in over to do this the data must be 16 Byte aligned. A number of vector datatypes are available depending on the needs of the data to make most efficient use of the 16 Bytes in each vector. The vector types segment the memory as follows:

- 16 X 8-bit int: vector (un)signed char
- 8 X 16-bit int: vector (un)signed short
- 4 X 32-bit int: vector (un)signed int
- 4 X float : vector float

When a DMA transfer occurs, the EA given for the transfer is the address of a 16 byte block in main memory. This 16 Bytes is transfered to the SPU as a single block of data. It is stored in the SPU as a 16 byte block which can be accessed as a vector. Even scalar values are stored and used as vectors on the SPU, so it is useful to combine multiple scalars into vectors and extract them as needed.

It is also possible to define custom structs that will fill the 16 Bytes of memory in a vector, or multiple vectors. This is usefull to transfer data, e.g. task parameters, to the SPU. For variables that do not use the entire 16 bytes it is necessary to create some padding data to fill up the remaining space.

The arrangement of data in memory will have a large effect on how efficiently data is loaded onto the SPUs for processing and hence the speed of the final application. It is most efficient to transfer data from continuous areas of main memory to the local store as maximum use can be made of each DMA transfer.

The image will be read into the application and stored using the RGB standard to represent the image data. This means that each pixel in the image has a red, green and blue component each requiring 8 bits(1 byte) of memory. Based on this the amount of memory required to store the image is :

```
imageSize = Width * Height * channels * sizeof(byte);
```

Figure 4.5: Data Layout in Memory

Because of the need to align data to 16 byte boundaries this is not strictly correct, and will depend on the way data is stored within the 16 byte blocks. Two methods for storing the image data in memory were examined, an array of structs containing pixel element with each of their Red, Green and Blue channels stored together, and a Structure of Arrays (SOA) which stores the image data in memory as seperate continuous arrays for each of the channels. Figure 4.5 illustrates how these would appear in memory.

The SOA method of storing data was chosen for this application due to a number of reasons described below. The first thing to examine is the restriction that data must be alligned to a 16 byte boundary. To make maximum use of memory and enable appropriate DMA transfer, we must pack a number of pixels into a single 16 byte space. If using the Array Of Structures (AOS) method it is possible to pack five pixels containing 3 Bytes of data each into a 16 byte block, however, 1 byte will be wasted within the data structure in order for it to be properly aligned. Alternatively, by using the SOA method for storing data we can pack 16 bytes of a single channel into the block of memory. This way we are efficiently using the entire space available and not wasting any memory on padding.

This brings us to the issue of granularity. When the data for multiple pixels is packed into a single block, it is necessary to load the entire block in order to access the data for just one. Similarly if some data is required on the SPU, the entire block must be transfered from main memory to the LS on the SPU.

The AOS method allows us to pack 5 pixels into a single block, so at worst we will be loading 16 bytes of data when all we require is 3 bytes. The SOA method of storing data means that it necessary to access 3 X 16 byte blocks of data to access the RGB

values of a pixel, leaving the potential worst case wasted transfer of 45 bytes when only 3 were needed. This overhead seems excessive, however, in reality if the application is designed well, the data will be operated on in with vector operations and single pixels reads and writes should be avoided.

This introduces the consideration of how the data will be operated on to create an efficient program. By loading the image into arrays of char data types it is possible to efficiently process this data. As described above, we can fit 16 8-bit ints into a single vector and it is possible to apply vector operands (also known as SIMD operands), which operate on all 16 elements of the vector in a single operation. SIMD operations are possible when data is structured in the SOA and AOS method, however, the extent that they may be used is specific to the application.

An example will highlight the difference between the two structures; in order to calculate the initial disparity cost it is necessary to get the absolute difference between two pixels for each channel and then multiply each channel by a weighting value to get a scalar absolute cost per pixel. With both the SOA and AOS methods it is possible to get the absolute difference between two pixels, returning a new vector of the corresponding values. With the AOS method this will be 5 RGB values and the SOA will be 3 vectors each containing a colour channel. The next stage requires each channel to be multiplied by the same value which is applied simply to the SOA vectors by splatting that value into a new vector and multiplying it by the correct channel. This operation is not so easily accomplished with the AOS data because the channels are all mixed in together.

Finally the SOA memory layout allows for easier DMA transfers to the SPU and has been shown to provide a significant speed-up compared to AOS code. Sung[29] compares the development of a 3D simulator using simple euler integration implemented with an AOS and SOA structure. The AOS structure takes 300ms to complete and the SOA structure takes only 80ms.

One reason for this speed-up is that SOA arranged data can be transfered in larger blocks. Regardless of how large a DMA transfer is, it will take a least eight cycles on the EIB, so the transfer should include the maximum amount of data which is 128 bytes[25]

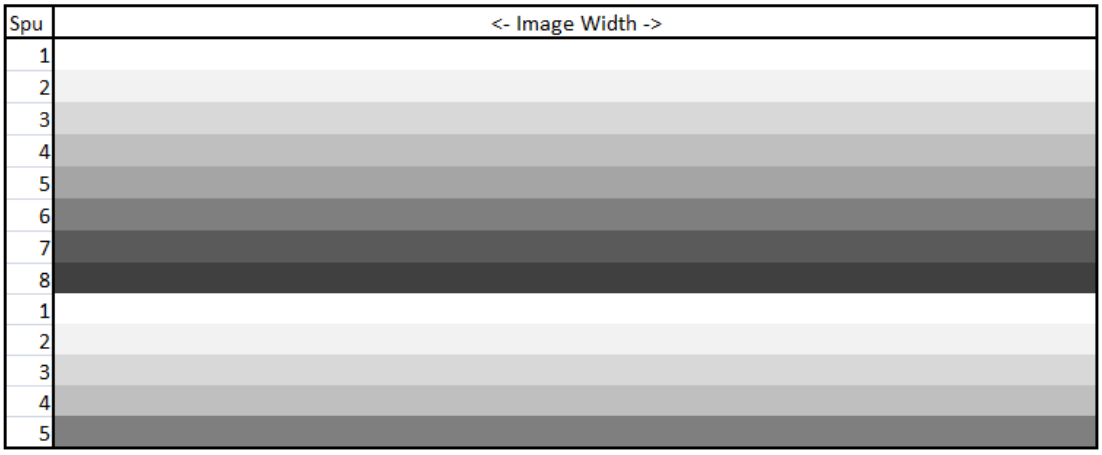


Figure 4.6: Cyclic Distribution

AOS data-packing often produces smaller code sizes, but it typically executes poorly and requires significant loop-unrolling to improve its efficiency[24]. In short the SOA structure allows SIMD operations to be programmed as if they are scalar operations, they fit well with the original algorithm and have a better performance.

Image Data Segmentation

The task of segmenting the image data to most efficiently use the SPUs depends on the requirements of the kernel being executed. Two segmentation methods are applied to the input images and disparity cost volumes which fit the needs of the algorithm being computed in the task kernel.

The first method as shown in fig 4.6 cuts the image into single lines and passes this data to the SPU for processing. This method is used by the initial cost calculation because each line can be processed in parallel by the initial cost algorithm. This method has the advantage that the task is heavily divided and can easily take advantage of all of the available SPUs to compute the task in parallel. Once an SPU is finished, the next available task is assigned to it without any need for specific ordering of data transfers.

The second method is used for the aggregation stage. The aggregation stage is not computed on a 1D line of image data, instead it requires a window around the target

pixel to be used for refining costs. This means that each SPU will need to hold data on a number of lines of both the image and disparity cost volume.

Considering a cost per pixel of 3 bytes, and a disparity volume storing 64 disparity levels per pixel each using 1 byte, we have a total memory usage of 67 bytes per pixel. If this were applied to a full HD image with a width of 1920 pixel a single row would consume 128640 bytes, and it would only be possible to store two rows of data before running out of memory on the LS.

As the example above shows, it is not possible to store sufficient data in the LS of a single SPU to compute the aggration stage by segmenting the image into full rows. The alternative method is to segment the image horizontally and dedicate a SPU to the calculation of this column.

Using the same figures as above with a column width of 64 pixels we get a memory usage of just 4288 bytes per row, allowing approximately 60 rows to be stored in the LS for computation. Figure 4.7 shows an image segmented into seven columns which are executed on seven SPUs.

There is a drawback to segmenting the images onto seperate SPUs which is caused by the data requirements of the aggregation stage. Just as the aggregation technique needs data from multiple rows to compute the aggregated cost, it needs data from its neighbouring horizontal pixels also. This means that some overlapping pixels need to be loaded for each column to calculate the correct values for pixels near the border of the column.

This is easily accomplished as the pixels are stored in blocks of 16. The adjacent blocks to the usable pixels are loaded and the usable pixels are then computed. Finally, the excess pixels are discarded instead of being transfered back to main memory. The need to load overlapping data adds an overhead that cannot be avoided to the SPU data. The actual cost per row for this method is calculated using:

```
rowSize = (usableBlocks + adjacentBlocks) * (pixelSize + disparitySize);
```

This gives the final cost per row for 64 usable pixels to be (64 + 32) * 67 = 6030 bytes. As can be seen, there is a considerable amount of overhead added by needing to transfer the additional overlapping data, but it is unavoidable due to the limited

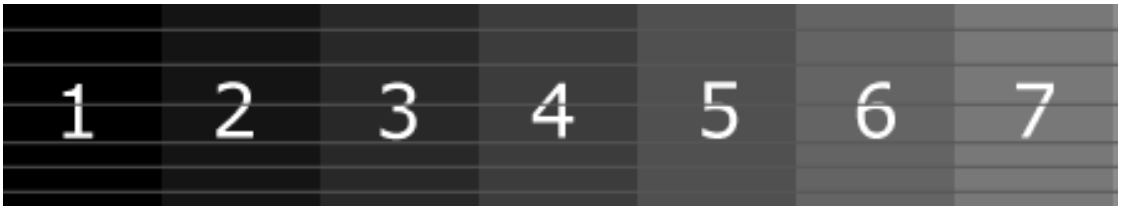


Figure 4.7: Block Distribution

memory on the SPU, the requirments of the algorithm, and the use of 16 pixel wide blocks to store the RGB channels.

The work blocks which segment the data on a line by line basis simply execute once per line, return the data to main memory, and then grab another line of image to process. Four way buffering is implemented to hide the cost of data transfers.

There is no state being maintained between each workblock since each workblock can be completed individually. This is not the case with the work blocks that execute on a column of data. This is due to the large amount of data needed to update the cost of a single pixel.

Consider the example of aggregating the cost of a single pixel with a window size of 15X15. The total number of pixels required to produce a result is 225 pixels. Considering the same cost per pixel as above, the data required per pixel is 15KB.

If the image was to be processed on a pixel by pixel basis, transfering all the data for a single pixel to the SPU for every pixel in a HD image would need in the region of 30GB of data transfered to the SPUs. This number is reduced by processing the data in rows and hence only having to transfer data for the new line which is being processed, however, there would still be a huge amount of redundant data being sent if the image is processed on a line by line basis in regions of 64 pixels wide.

The solution to this problem is to use multi use workblocks which can maintain their state in the memory defined by the ALF runtime as the task context. In order to store enough data the Task context is defined to act as a buffer for image data so that for each iteration of the work block we must simply load a single line of pixels and disparity. This reduces the total transfer per column to be :

```
total transfer = rowSize * (numRows + windowSize)
```

The reason we must add the window size to the calculation is that the first few work block iterations will simply involve filling the buffer with data. Additional padding is applied to the borders of columns which are at the edge of the image and at the top and bottom to signify a maximum cost in those regions since they do not exist in the original image.

The buffer on the SPU is operated as a FIFO buffer with new data pushing the oldest data out of the buffer and being discarded. This method significantly reduces the amount of data which needs to be transfered, and allows the solution to scale succesfully to much larger images by removing the link between image width and memory usage on the SPU. In fact, this method makes the memory usage completely predictable based on the number of disparity layers, window size, and the number of pixels blocks being used as described in the equations above.

This method of image segmentation can also use the highlest level of ALF runtime buffering techniques because although the buffer is large, the input and output buffers are both relativly small.

This method of data segmentation and multi task execution is based on the streaming model. Data is "streamed" through the SPE operated on by a single kernel and returned to main memory. The data transfer is completed while further computation is taking place.

Data Transfers & ALF Buffering

The low level details of data transfers are managed by the ALF runtime, however, it is still necessary for the code to describe exactly what data needs to be transfered and when. This is done on the SPU side by defining a function which is called by the ALF runtime for each work block. This function corresponds to the "Prepare input DTL" function in figure 4.2. In the case of multi use work blocks, the "Prepare input DTL" function is called for each iteration.

The function creates the DMA transfer lists which will transfer the data that is needed for the corresponding compute kernel, and will be transfered to the SPU LS by the ALF runtime using the optimum buffering technique as illustrated by figure 4.8.

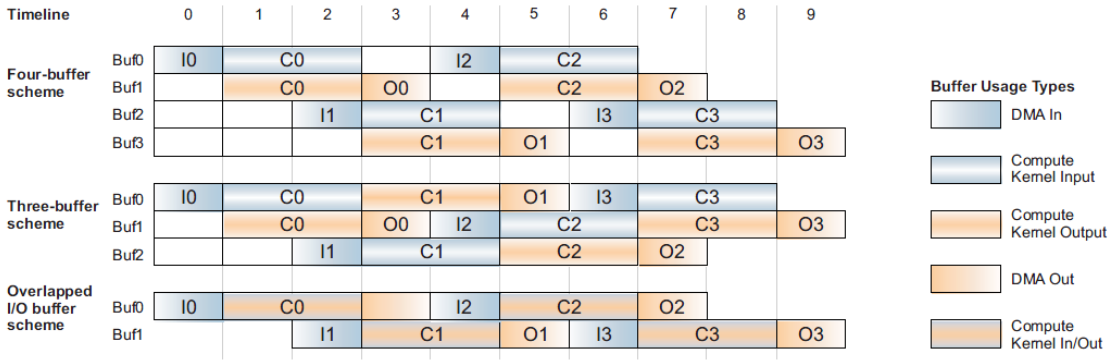


Figure 4.8: ALF Buffering Techniques

The four way buffering scheme can be used if there is enough space in the SPU to store two input and two output buffers, and if the overlapping IO buffer is not used. This is the optimum solution and all data transfers in this implementation are designed to allow four way buffering to operate to efficiently transfer the data to the SPU.

In order to create the DTLs on the SPU, we require the EA address in main memory of the data required. To pass this address to the SPU DTL function a struct common to both the PPU and SPU code is used. This struct is passed to the workblock during its creation and contains parameters for the specific work block. Multiple EA addreses may be passed to a single work block depending on the needs of the function.

In the case of a multi-use workblock the SPU must calculate the address of the next block of data without further interaction with the PPU managment code. This is done by incrementing either by the size of data transfered or by the width of the image, depending on the way the image is segmented.

Data Transfer Lists (DTL) are created using builtin ALF API calls, for example :

```
ALF_ACCEL_DTL_BEGIN(dtl, ALF_BUF_IN, 0);
ALF_ACCEL_DTL_ENTRY_ADD(dtl, 100, ALF_DATA_FLOAT, host_addr_1);
ALF_ACCEL_DTL_ENTRY_ADD (dtl, 100, ALF_DATA_FLOAT, host_addr_2);
ALF_ACCEL_DTL_END(dtl);
```

This creates a DTL to the input buffer on the SPU. The actual transfers are described by the two middle lines, where dtl is the id of the DTL, 100 is the number

of 16 byte blocks to retrieve, ALF_DATA_FLOAT is the datatype being recieved and host_addr_1 is the EA of the data in main memory.

Each stage of the pipeline being implemented requires a different set of DTLs to be created, for example the initial cost computation requires one DTL for each channel of image data from both the reference image and the target image.

Output DTLs follow a similar format except they specify a ALF_BUF_OUT as the type which tells the ALF runtime to read from the SPU LS which is dedicated to the output buffer.

4.6 Task Implementation

The following sections will describe the details specific to the implementation of each modular component, describing how each stage was developed to utilize the ALF runtime described in section 4.3

Image Loading

Loading the images into memory is the first step in the implementation, and like all other functions in the modular design this step is created as a seperate function. This allows for easy expansion of the framework to support an alternative file formats or video by loading it on a frame by frame basis into memory.

The process of loading the image is implemented on the PPU. The function is passed the name of the file to be loaded and also the address in memory where it is to be stored. The function then outputs the image in the SOA format described in section 4.5 and illustrated in fig 4.5.

The image loading function reads image files in the Portable Pixel Map (PPM) format, which is a simple file format that contains a binary list of integers representing the Red Green and Blue channels of the image. The data is not in the format required by the application so must be converted to fit the requirements of the framework.

The first step is to parse the image height and width which are stored at the beginning of the file. This can be achieved using a simple file read command in C.

```
fscanf (imageFile, "P6\n%d %d\n255\n", &width, &height);
```

Next the image is stored in the correct address in memory which has been allocated with the correct size and memory alignment by the initialization code. The pixels are being read in one at a time, so it is necessary to access the elements that make up the 16 byte vector individually. This is achieved by using a union type which allows the data to be accessed as both a vector type and an array of elements. The vector is accesed by using pixelVal.vec and individual chars can be accessed by using pixelVal.iVal[x].

```
typedef union _pixel_t
{
vector unsigned char vec;
char iVal[16];
}pixel;
```

This function is used to load both the reference image and the target image into memory.

Cost Initialization

The cost initialization is implemented as a parallel task using the ALF framework. The task is created by segmenting the image into individual lines as decribed above and executed on the SPUs in a cyclic distriution as shown in fig 4.6. The number of tasks created will be equal to the number of rows in the image.

The exact scheduling of the tasks is handled by the ALF runtime and there is no gaurantee that the blocks will be executed in order. For this reason it is necessary to ensure that this task is complete before the cost aggregation stage is started. This is accomplished by creating an ALF dependency between the task. This is done by passing the task handle to the appropriate ALF API:

```
alf_task_depends_on(aggregation_task_handle, iCost_task_handle);
```

Task specific details such as data transfer sizes and the number of disparity layers to evaluate are passed to the task as it is created, and are automatically transfered to the SPU upon execution of each workblock by the ALF runtime component.

The cost initialization compute kernel implements the algorithm described in section 3.7 as follows: We must compare each pixel in the reference row to X number of pixels in the target image. The value of X is the number of disparity levels which are the be evaluated. We compute the disparity using vector unsigned chars which compute the initial cost 16 disparity levels at a time. This takes full advantage of the vector operations available on the SPU.

In order to compare the target pixel to each of the pixel vectors in the reference image, we must fill a vector with the value of the reference pixel. This is done using the spu intrinsic command spu_splats which initializes a new vector whose elements all equal the scalar value.

```
refR = spu_splats(*referencePixel);
```

This is repeated for each colour channel for the reference pixel. The absolute difference between the reference pixel is then computed using SIMD maths operations for each level of disparity.

Once the absolute difference has been calculated, the dot product between the pixel RGB values and a greyscale vector is computed to return a single value representing the disparity cost.

Finally, this value is truncated and scaled to fit inside the 8-bit integer bounds and saved into the disparity vector in the output buffer.

The ALF framework then calls the function which transfers the data from the output buffer back to main memory.

Cost Aggregation

Two methods of cost aggregation have been developed; Square window and Adaptive window which implement the algorithms described in the section 3.8. Both methods are implemented as seperate compute kernels, however, they share the same IO functions as defined by the modular framework being developed.

Both aggregation functions also share the same streaming compute kernel model which required the development of a buffering system to save data on the SPU between tasks to reduce the data transfers required.

The basic structure of both kernels are implemented as follows:

A buffer is created using the task context. A structure is defined which contains a one dimensional array for each channel of the reference image, and a two dimensional array which stores the disparity cost for each pixel. The task context also maintains two variables which contain the index of the oldest value within the buffer and also the central row of the buffer. The index of the oldest value within the buffer corresponds to the top left pixel within the buffer as the workblocks progress down the image column.

On each iteration of the workblock, a single line of data is loaded from main memory to the input buffer of the SPU. This data is copied into the aggregation buffer stored in the task context and each of the buffers indexes are incremented to represent the new data.

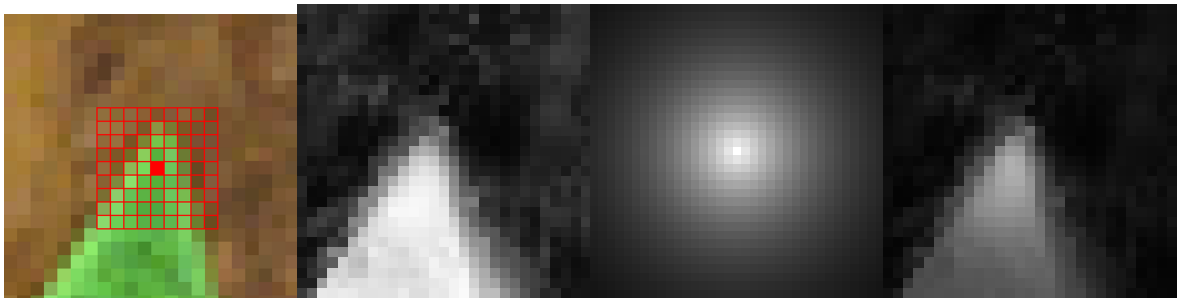
The second index for the buffer corresponds to the current row being operated on by the compute kernel. Once the data has been loaded into the buffers, the compute kernel executes either the square window or adaptive weight algorithm depending on which compute kernel was selected when creating the task.

Square Window The square window compute kernel simply refines the cost of each pixel by summing the costs of all neighbouring pixels within the window size. This calculation is done by summing the values stored for each 2D window of disparity space, and then scaling the result to fit into the 8-bit integer return value.

The disparity cost volume stores values as vector unsigned chars so in order to calculate the sum of their results we must convert them to vector unsigned shorts to ensure data is not lost by overflowing the variable storing the result.

The disparity costs are converted to vector unsigned shorts by using a vector shuffle intrinsic. This takes the upper or lower half of the values and stores them in the lower 8-bits of the space for each short inside the vector while filling the upper 8-bits with zeros. This conversion is possible because all integer value are stored in vectors in a similar format.

Once the sum has been computed, the value stored is divided by a scaling value, and then compared with 255 (the maximum value of an 8-bit int) using `spu_cmpgt(x, y)`. This function returns a bitmask for the corresponding values which are larger and smaller than 255. If the value is smaller we shuffle it back into its 8-bit char result



(a) (b) (c) (d)

Figure 4.9: Adaptive Weight Window Calculation

vector, if it is larger we simply put the value of 255 in the result.

Adaptive Weight The adaptive weight calculation requires more steps than the square window method, however, it follows the same basic principle. Figure 4.9 shows the stages used to create the weights.

First the weights to be applied to each pixel in the window need to be generated. This is accomplished by comparing the reference pixel with each of the other pixels within the window, and calculating the absolute difference as decribed in the initial cost calculation. The result of this is a float value for each pixel in the window which contains the value of the weight that pixel has to contribute based on the pixel similarity principle. This is illustrated in figure 4.9 (b). The brighter a pixel is, the larger a weight it will contribute.

The next stage is to combine the similarity weight with the proximity weight (fig 4.9 (c)). The proximity weight does not need to be recomputed for each pixel as it is based on a pre-configured value, as such this can simply be stored in a seperate area of the task context and retrieved for each pixel. The two weights are multiplied together using SIMD floating point multiplication to produce the final weight for each pixel (fig 4.9 (d)).

Next, the disparity cost of each pixel within the window is multiplied by its weight and added to the total sum for the reference pixel. This calculation is done at four disparity levels simultaneously.

Finally, the disparity cost is passed to the disparity computation stage which determines the lowest cost disparity level for the reference pixel.

Disparity Computation

The disparity computation stage of the algorithm implements the WTA algorithm as described in section 3.9 of the design section. This algorithm simply searches the disparity space for each pixel in the image and finds the location of the lowest cost within that row of data.

The actual implementation is optimised for SIMD operations, which significantly reduces the number of iterations required to determine the lowest cost. This is accomplished by first reducing the seperate disparity vectors against each other and selecting the lowest value from each iteration.

The comparison is done using the `spu_cmpgt` intrinsic which compares two vectors and returns a vector of the same order. This vector contains a bit mask for each value containing zeros or ones depending on if the value at that location is larger of smaller. The bit mask is then applied to the two vectors to extract the lower values from each location in the vector and returns a new vector. We must also store the index of the disparity level with the lower value so we use the same mask on vectors containing the index of the appropriate vector.

This process is repeated until only one vector remains which is then iterated over to find the lowest value. This value is then scaled if necessary and stored in the output buffer.

The disparity computation stage of the implementation was originally designed to be a seperate modular component, however, during implementation and testing the overhead associated with re-loading the entire disparity cost volume for such a small amount of computation was deemed to be too high, and as a result the disparity cost function was combined as a final phase of the cost aggregation stage.

Disparity Refinement

The disparity refinement stage is a post processing filter that is applied to the disparity map the remove small pixel errors. The task depends on the stereo aggregation stage (which outputs the disparity map) and cannot begin operation until the that task has completely finished.

The task uses the block image segmentation method described above to process the image because it requires 2D areas of the image to operate on. This instance of the block segmenation is only applied to the disparity map which is a single channel (greyscale) image, as such there is only a limited amount of data transfer associated with this stage.

The compute kernel applies a 3X3 median filter to each pixel of the image. The median filter is computed by comparing each pixel in the 3X3 window surrounding the pixel, sorting those pixels into their assending order and returning the median value.

This process removes many incorrectly identified pixels while maintaining the sharpness of the image.

4.7 Optimization of Implementation

In order to exploit to power of the cell processor it is necessary to optimize the code it will process. This includes avoiding branches in the code, loop unbundling, and correct data managment as described above. Altough a limited amount of optimization is completed by the compiler, it is necessary for the programmer to write code which will perform well on the cell platform.

Algorithm Optimization

Although branches are generally to be avoided in on the SPU, it is sometimes worth adding branch code when it is potentially possible to skip a large number of instructions. One such possibility exists within the adaptive weight algorithm that has been developed. When the similarity weight of a pixel has been computed a simple if statement is evaluated to check if this value is smaller than zero. If this is true, then that pixel will not contribute to the final result so we may skip the remaining code for that pixel and continue with the next. This process will remove approximately 40% of pixels from being computed, depending on the change of intensity over the regions. Under testing this was shown to provide an increase of 18% to the overall speed of the algorithm.

SIMD Intrinsics and Scalars

The use of SIMD instruction is essential to take full advantage of the SPU processing speed. The cell processor is a vector processor and is designed to operate on multiple values at one time. Traditionally the programmer has had to rely on the compiler to optimize code to take advantage of SIMD operations, however, the cell libraries provide the SIMD intrinsics which allow the programmer to directly work with low level SIMD operation. The SIMD intrinsics are essentially inline assembly with C function call syntax [30].

The SPE only accesses its LS a quad-word (16 byte vector) at a time. The use of scalars variables requires extra operations to rotate a scalar value into a specific location in a vector before it can be processed on the SPU. To efficiently use scalar values on the SPU they should be packed into a vector value. This can be done by simply changing the scalar to a vector value which wastes space, or by packing a number of scalars into a vector.

Branch Prediction

If at all possible it is important to avoid branches because if the wrong branch is predicted, the processor will stall for 18 cycles [30] while it loads the correct instructions for the branch.

It is not always possible to completely remove branches from code, and there are certain circumstances when a branch may improve the speed of an application by skipping a large chunk of redundant operations. When branches cannot be avoided, it is important to specify to the compiler which branch is most likely to be taken to avoid badly predicted brances. This is achieved by using the language extension `__builtin_expect((a > b),0)`. This code directs the compiler that the more likely result of the if statement is false.

A good example of how large an effect this can have, during optimization an if statement was placed to cull excess operations. When the incorrect branch is given precedence, execution time for the test was 11.3 seconds. However, when the branch predicition is correct, prioritising the branch taken more often, execution time is reduced to 7.46 seconds, an impressive 33% improvement in speed.

This simple example shows how a single change can have a large effect on performance on the cell platform. This large change in performance is due to the fact that the SPU does not contain branch prediction instructions. Each branch instruction generates a target address in memory containing the instruction sequence for the branch. If the branch is taken, the processor will execute instructions at this address instead of the next address in the sequence. The SPU always predicts that the lesser of two addresses will be taken and fetches instructions before they are used. If the SPU grabs the wrong instructions program execution will stall.[25]

Loop Unrolling

Loops are a key requirement of many applications, this is especially true for image processing application when it is necessary to perform an operation on each pixel in the image. The problem with loops on the SPU is that on each iteration a branch expression has to be evaluated to decide whether another step is required. Programs containing multiple deep loops will perform particularly badly on the SPU because of constant stalls as the loop exits.

The method of avoiding this penalty is to unroll loops. This simply involves flattening out the loop by manually copying the code multiple times. It is possible to completely unroll and remove loops when there is a constant number of iterations. When this is not possible it may still be possible to perform a number of updates within a single loop and hence reduce the number of iterations required.

This process of manually unrolling loops can provide a significant performance boost. During optimization a single level of a loop for the aggregation stage was unrolled which lead to a 15% increase in calculation speed. By further expanding this unrolling to remove a second level, a further 4% increase was achieved.

4.8 Summary

This section has described in detail the methods used to implements the stereo vision algorithms. It has described the ALF framework and how that is used to build a portable application capable of running on a number of cell platforms.

The topic of memory management has been discussed and a number of issues have been described which affect to overall design of the implementation. Finally the implementation details of each stage of the pipeline was described and a number of optimization stategies were introduced.

Chapter 5

Evaluation

This section of the paper evaluates the performance of the implementation in two ways. First, the quality of the disparity map created is evaluated. This evaluation compares the square window and adaptive weight algorithms implemented with different window sizes to examine the effect this has on the final result.

Second, the performance of the implementation is examined by testing the speed of the algorithm with respect to changes in resolution, number of SPUs used and finally against an x86 implementation. The purpose of this wide range of tests is to evaluate how well the cell processor performs as an image processing platform rather than simply to evaluate the specific stereo algorithm that has been implemented. The direct comparison to the scalar x86 code shows how efficient the cell platform is for this type of work.

Each section will first explain the motivation and details of the test being implemented and then describe the results gained from the tests.

5.1 Aggregation Techniques

In order to evaluate the quality of the disparity maps created, it is necessary to know the true value for the disparity of each pixel in the test image. The Middlebury stereo dataset [6] was used as a means to evaluate the implementation because this dataset includes a ground truth disparity map for each stereo image set. The middlebury

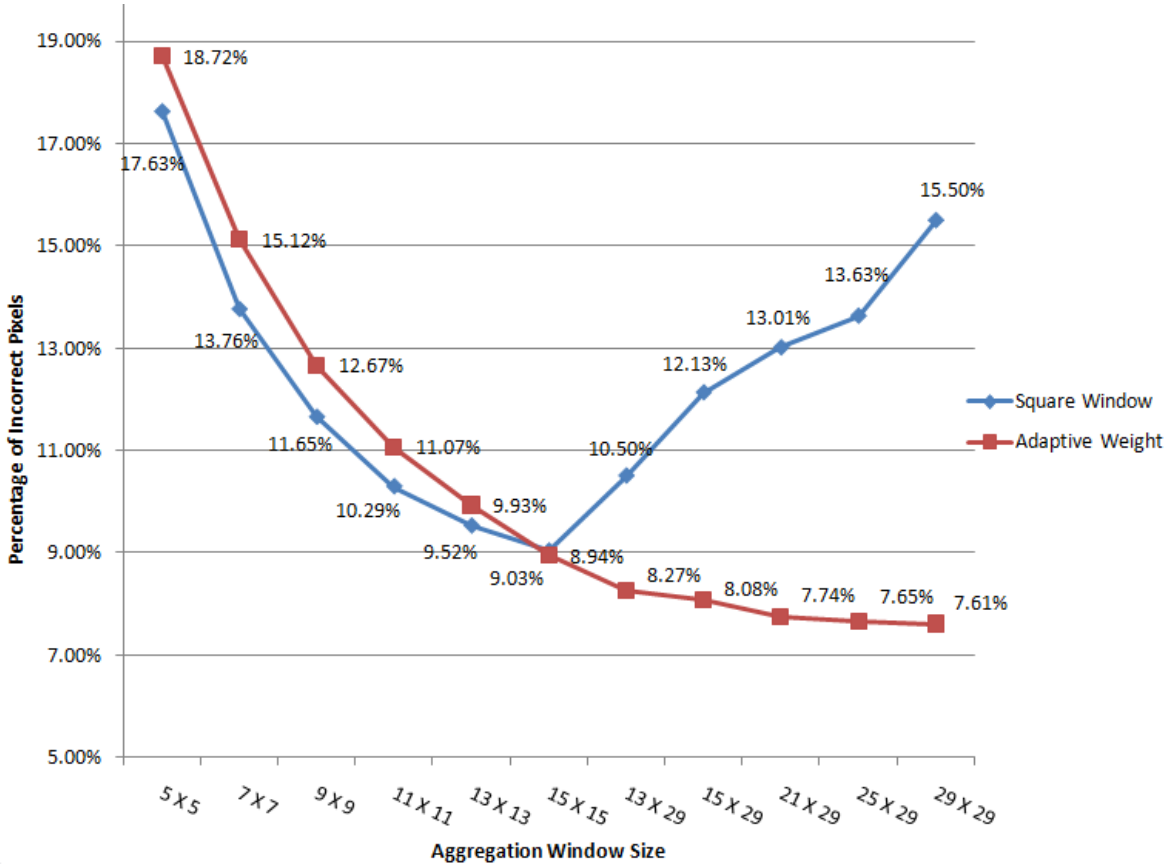


Figure 5.1: Disparity map quality

university stereo vision web page also contains a list of rankings for many state of the art stereo algorithms which allows for an objective comparison of the implementation against these methods.

Disparity maps are evaluated based on the number of incorrect pixels in the image. A pixel is deemed to be incorrect if the difference between the ground truth pixel and the disparity pixel is larger than a set threshold.

Both the square window and adaptive weight aggregation algorithms have parameters which control the size of the support window that is used. The first experiment examines the effect of adjusting this parameter has on the quality of the disparity map for each algorithm. Figure 5.1 illustrates the results obtained for support window sizes ranging from 5X5 to 29X29. The cones image shown in figure 5.2 from the middlebury dataset was used for this test.

The following can be observed from the results:

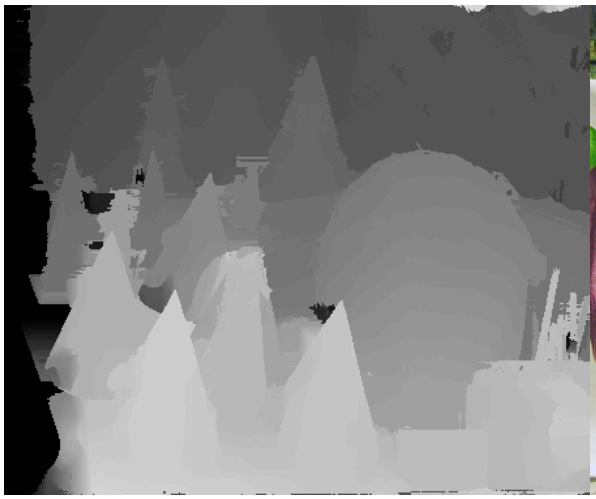
- The error rate is high when a small support window is used. As the support window size increases, the overall quality of the disparity map increases. This is expected as a feature of the support window is to spread the cost over a larger area and thereby reducing the effect of noise in the initial disparity cost.

- The square window algorithm has a peak performance with a window size of 15X15 after which the quality of the disparity map begins the degrade quickly. Increasing the size of the support window continues to increase the smoothness of the disparity map, however, when the support window is applied to an area including a depth discontinuity, the incorrect value is produced because pixels from regions which are not at the same disparity level contribute to the result. The overall effect is a blurring around the borders of objects. Figure 5.3 shows this effect by comparing a section of the ground truth with the disparity map created with a window size of 29X29. The regions marked in red are those which are incorrectly matched and can be seen around the borders of the objects.

- The adaptive weight algorithm performs worse than the square window algorithm for the first six window sizes evaluated. This is despite the fact that the algorithm is much more complex and compute intensive. The reason for this is

(a) Cones Image

(b) Teddy Image



(c) Cones Disparity Map

(d) Teddy Image

Figure 5.2: Middlebury Dataset Images

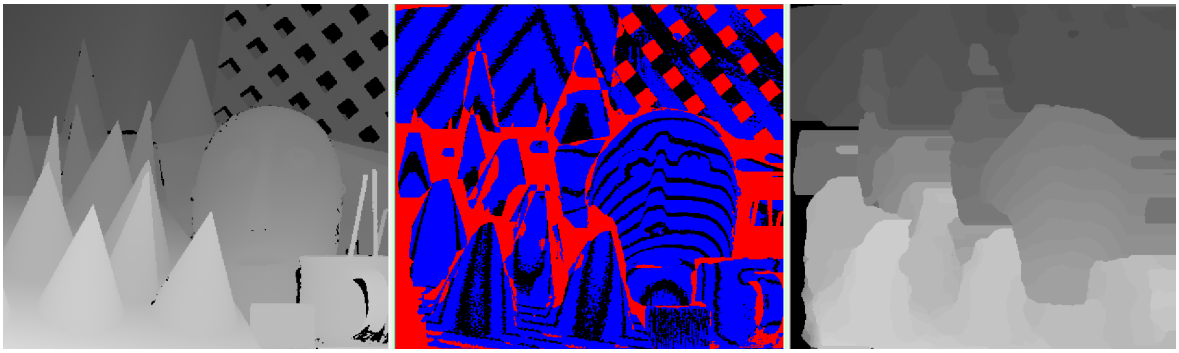


Figure 5.3: Comparison with ground truth(left), difference(center) and 29X29 square window(right)

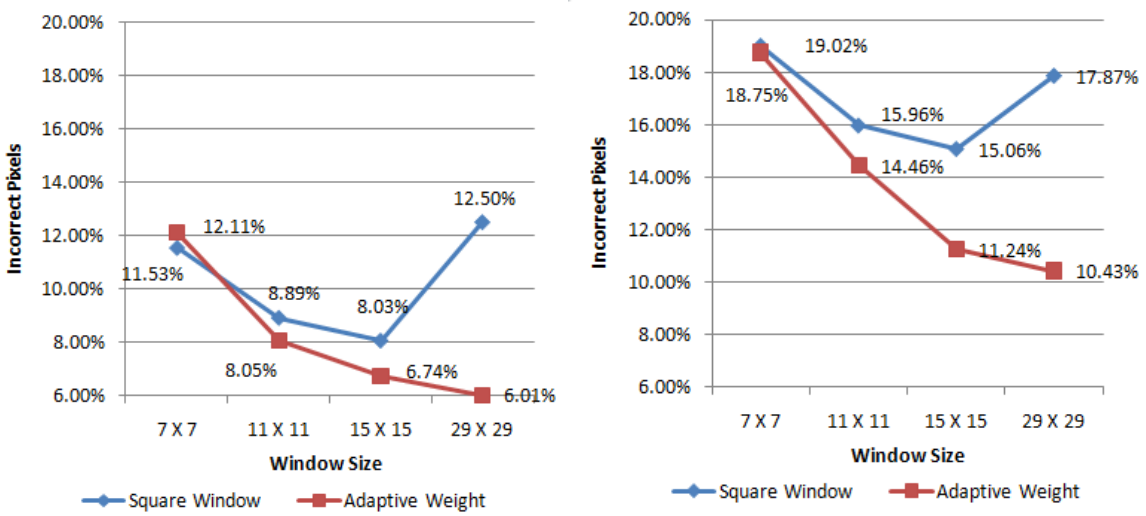
that the algorithm actually reduces the number of pixels that will contribute to the aggregated cost hence reducing the signal to noise ratio which is important for a window based approach. The conclusion to be made from this is that the adaptive weight is not worth calculating for window sizes under 15X15 because the performance is matched or bettered by the simpler implementation of the square window.

- The results for the adaptive weight algorithm continue to improve as the window size increases, however, the relative gain in quality begins to diminish as the size increases.

- Compared to the original implementation of the adaptive weight algorithm by Yoon and Kweon [7], this implementation does not achieve the same quality disparity map. The disparity map obtained by Yoon and Kweon for the cones image set contains only 3.97% incorrect pixels [6].

  The window size of the current implementation is one factor contributing to the reduced quality. This is restricted to a size of 29X29 due to the limited amount of memory available on the SPU. Although it would be possible to process less pixels per SPU and thereby increase the usable window size this would decrease the overall performance.

  Other factors that reduce the quality of the disparity map produced are the simplifications made during the design of this project. That is, this implementation only calculates the similarity weights once for each pixel and only process the



(a) Tsukuba Image

(b) Teddy Image

Figure 5.4: Middlebury Dataset

image from the point of view of a single image. This introduces pixels which cannot successfully be matched near occluded areas.

- When compared with the openCV implementation of a stereo algorithm the results are favourable. The openCV algorithm produces a disparity map with an average error of 13% bad pixels for the cones image. Both the square window and Adaptive weight algorithms produce considerably lower error rates.

Further evaluations done using the Middlebury dataset can be seen in figure 5.4, which evaluates the Tsukuba and the teddy images. The results from these datasets confirm what is observed in the cones image; that the square window approach only improves until a certain point and then quality begins to reduce, and also that the adaptive weight continues to improve. Similar to the cones image, the adaptive weight algorithm does not perform as well for these two image sets as the original implementation.

The variation in the overall incorrect number of pixels obtained between the different sets of images can be explained by the complexity of the images and the number of disparity levels which are evaluated. The Tsukaba image set only contains 16 levels of disparity and is a lower resolutions image than both the cones and teddy images.

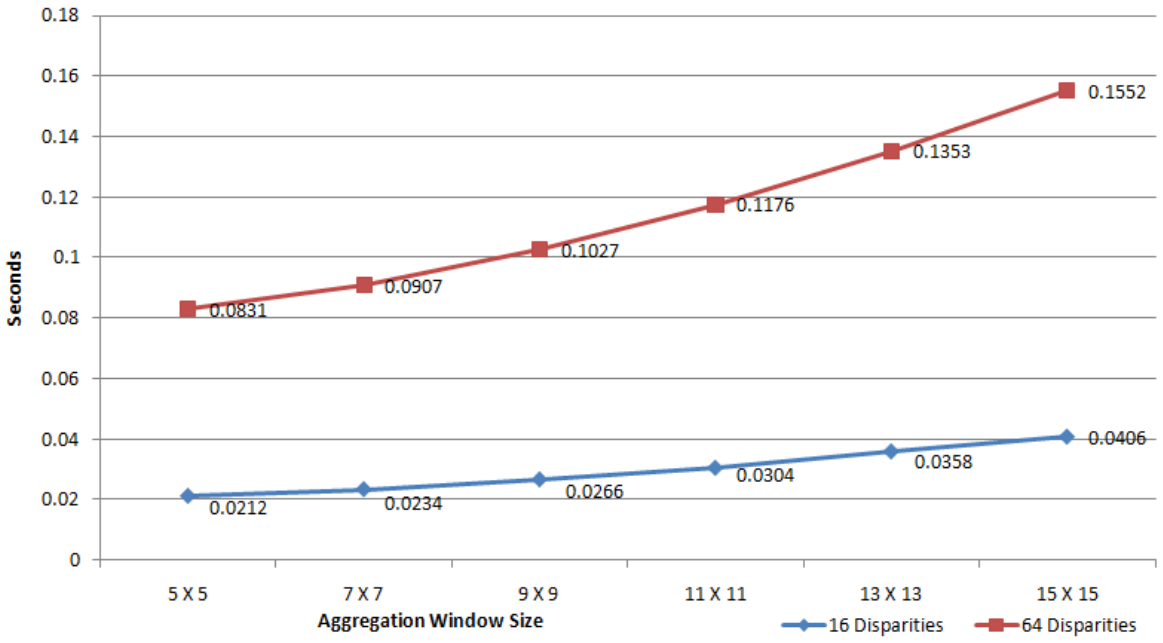


Figure 5.5: Square Window Timing Evaluation

The highest number of incorrect pixels were detected in the Teddy image, and this can be attributed to the complexity of the scene. As shown in figure 5.2, there is a large number of curved and similar surfaces which cause incorrect results in the final disparity map.

The results obtained follow the general trend of results obtained by algorithms tested on the Middlebury dataset. That is the Tsukuba image generally has the lowest number of errors, the cones has a medium number, and the teddy image generally has the highest number of incorrect pixels.

Overall, the adaptive weight algorithm is capable of producing a higher quality disparity map than the square window approach as the window size increases. The next section will evaluate the speed of both algorithms with regard to window size.

5.2 Speed verses Window Size

The previous experiment has shown that the size of the support window is important for the overall quality of the disparity map that is produced, and a larger support

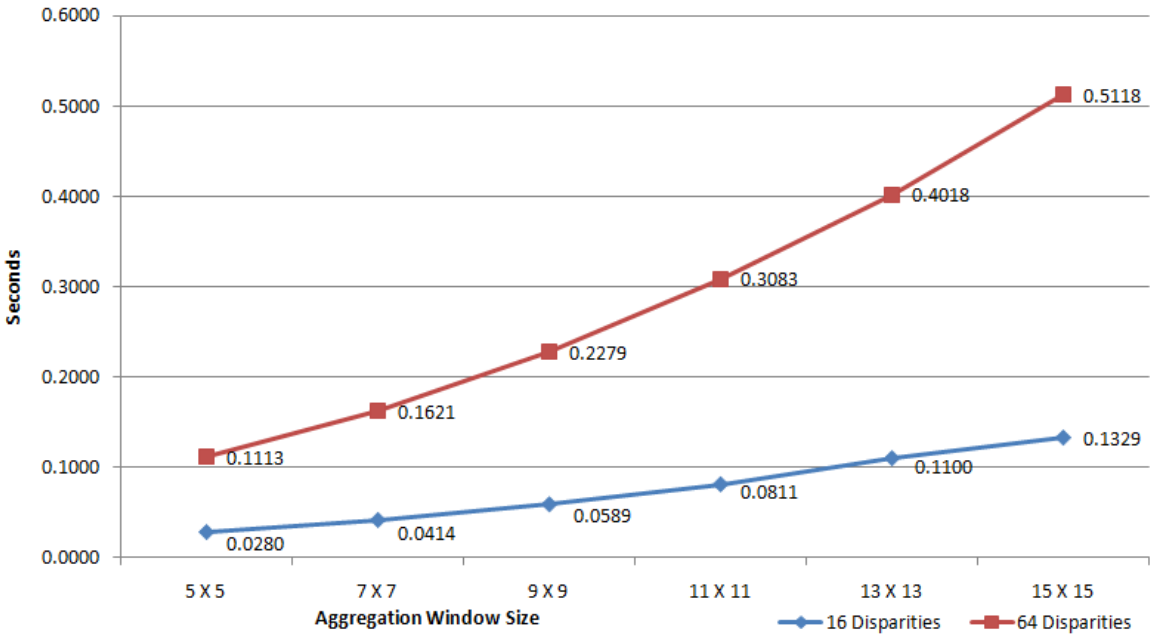


Figure 5.6: Adaptive Weight Timing Evaluation

window will generally produce a higher quality result. The problem is that a larger support window requires more computational resources to create the disparity map, and if the disparity map is needed in real time, or near real time, it is necessary to trade quality for speed.

The speed of each algorithm is evaluated as follows: a fixed size image with a resolution of 450X375 is used for the experiment. The image is evaluated with 16 and 64 disparity levels, and the total time for completion of the disparity map is recorded. This process is repeated for both aggregation algorithms with a range of window sizes from 5X5 to 15X15.

The results of the experiment for the square window method are plotted in figure 5.5 and the results for the adaptive weight algorithm are plotted in figure 5.6

The following can be observed :

- Assuming an acceptable real time speed is 24 fps, the time available to compute the disparity map for the algorithm to be considered real time is 0.0416 seconds. It can be observed that the square window method is capable of evaluating 16

disparities in real time with all the window sizes evaluated. The adaptive weight can operate in real time only up to a window size of 7X7.

Based on the quality results obtained in the first experiment it can be seen that the quality of the adaptive weight algorithm is equaled or surpassed by the square window method for window sizes this small. Therefore it is clear that the adaptive weight algorithm cannot be considered usable in real time applications.

- The time taken to complete the calculation increases linearly based on the number of disparity maps. This is observed to be true for both the adaptive weight and square window algorithms, and is due to the direct link between the number of disparity levels required and the amount of processing which must be completed per pixel.

- Neither algorithm is capable of true real time results with 64 disparity levels. The square window method is able to produce a result of 12 fps with a 5X5 support window or 6 fps for a 15X15. The time required to move the disparity cost volume to and from the SPU memory is the limiting factor for the square window algorithm. This data transfer cost is hidden by the buffering techniques during the adaptive weight calculation, which is much more compute intensive.

- The square window method scales much better than the adaptive weight algorithm. As the window size increases from 5X5 which contains 25 pixels to 15X15 which contains 225 pixels, the time to compute the completed disparity map only doubles. The same change in window size for the adaptive weight algorithm causes a 5 times increase in processing time.

As these figures show, the adaptive weight algorithm is a compute intensive task, and the time required to process even a small image increases rapidly as the size of the support window is increased. The slow speeds of the algorithm even at low resolutions means that it is not suitable for real time applications without considerable optimization.

It is important to note that this low resolution image does not allow for the use of all 16 SPUs available. This is because of the width of the columns used to segment the image for the aggregation stage. It is possible to optimize the column width to take

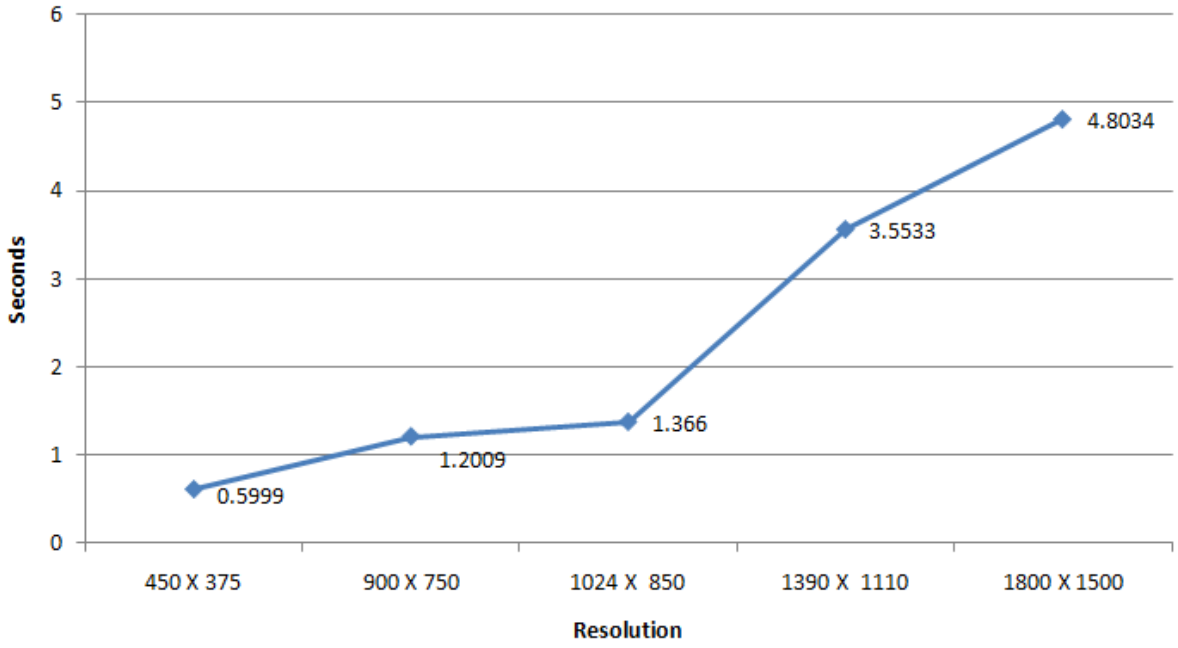


Figure 5.7: Computation time for different resolutions

advantage of all SPUs and improve the speed of the algorithm but there will be a large amount of redundant data transfers if the image was to be over segmented. The issue of column width will be discussed further in the next section.

5.3 Speed verses Resolution

The next stage of the evaluation will test the effect of different resolution images on the running time of the implementation. The primary reason for this is to highlight the effects of the segmentation of the images at different resolutions.

The test is conducted by running the adaptive weight algorithm on image sets with resolutions ranging from 450X375 to 1800X1500. The image for each test is the Middlebury cones image and all other settings remain the same to ensure that the only variable is the resolution of the image.

The results of the test can be seen in figure 5.7. A number of irregularities can be observed which are caused by the segmentation of the image into columns as shown in

figure 4.7. This segmentation is necessary for the aggregation stage of the algorithm to function properly, however, it imposes a number of restrictions on the application, most importantly, how the algorithm makes use of the SPUs.

The issues that can be observed are:

- First, the time taken to process the second image is only double that of the first, even though the area of the image is four times that of the first. This is caused because the first two resolutions cannot successfully use all 16 available SPUs to compute the cost aggregation stage of the process. The column width is fixed at 64 pixels (4 * 16-pixel vectors). This means for the image with a width of 450 pixels, only 7 SPUs are used to calculate the aggregation stage, compared to the image with a width of 900 pixels where 15 SPUs are used to calculate the aggregation stage. The reason the processing time is only doubled for the second image is simply because the number of rows in the image is doubled.

- Second, there is a 260% increase in processing time between the 1024X850 image and the 1390X1110 image even though the change in surface area only includes an increase of 170%. This is caused by a similar reason to the point above, however, the exact situation is slightly different. For this resolution, the width of the image is large enough to make use of all of the SPUs `1024 / 64 = 16 SPUs`. Examining the second image, the number of SPUs required is - `1390 / 64 = 21`. What is occuring to cause the large increase in time is that the first 16 columns are executed in a similar time to the smaller resolution image (with the added time of the additional rows), but once this processing is complete, a further 5 columns must be computed which presents the same problem as discussed above. This will leave a number of SPUs idle while waiting for the task to complete.

- The optimal situation is one where the image width is equal to the width of the columns multiplied by the number of SPUs available to process the task. In the case of the test data this is true for the 1024X850 image, which is why there is a dip in the graph for this resolution.

These results highlight the need to properly assign the width of each column based on the width of the image and the number of available SPUs. The simple equation

`ColumnWidth = ImageWidth / SPUs` would seem to provide the solution to this problem, however, as was briefly mentioned during implementation, there are a number of restrictions imposed on the column width: details of the SPUs available memory, memory alignment, and specific needs of the algorithm.

The following list will briefly summarise these restriction:

- The column width must be a multiple of 16 pixels. This is because 16 pixel values are packed into a single pixel vector.
- The window size must be padded with 16 pixels on either side of the usable data. This is required by the window based aggregation technique and introduces a large overhead if the number of usable pixels in the column is too small. For example, if a column width of only 16 usable pixels was chosen to maximize the spread of work across the SPUs, there would be 32 pixels of overlapping data being loaded for each calculation.
- The maximum memory available on the SPU for the column buffer is approximately 150KB after leaving space for the object code and also enough space for the input and output buffers needed to transfer data back to main memory.
- The memory cost per pixel is 3 bytes (colour pixel) plus 1 byte per disparity layer. In the case of an image with 64 disparity layers, we can have `150 * 1024 / 67 = 2292` pixels in the buffer at one time. Depending on the number of rows required, this imposes a restriction on the maximum width of the column.

As this shows, there are a number of issues to address when choosing the column width which make it difficult to produce a general solution that will be optimal for all resolutions and number of SPUs. The algorithm will achieve its best results if it is tuned to a specific resolution.

5.4 Effects of Modular framework

The use of a modular framework has a definite increase in memory transfer because as each stage completes, the data is stored back to main memory from the SPUs LS.

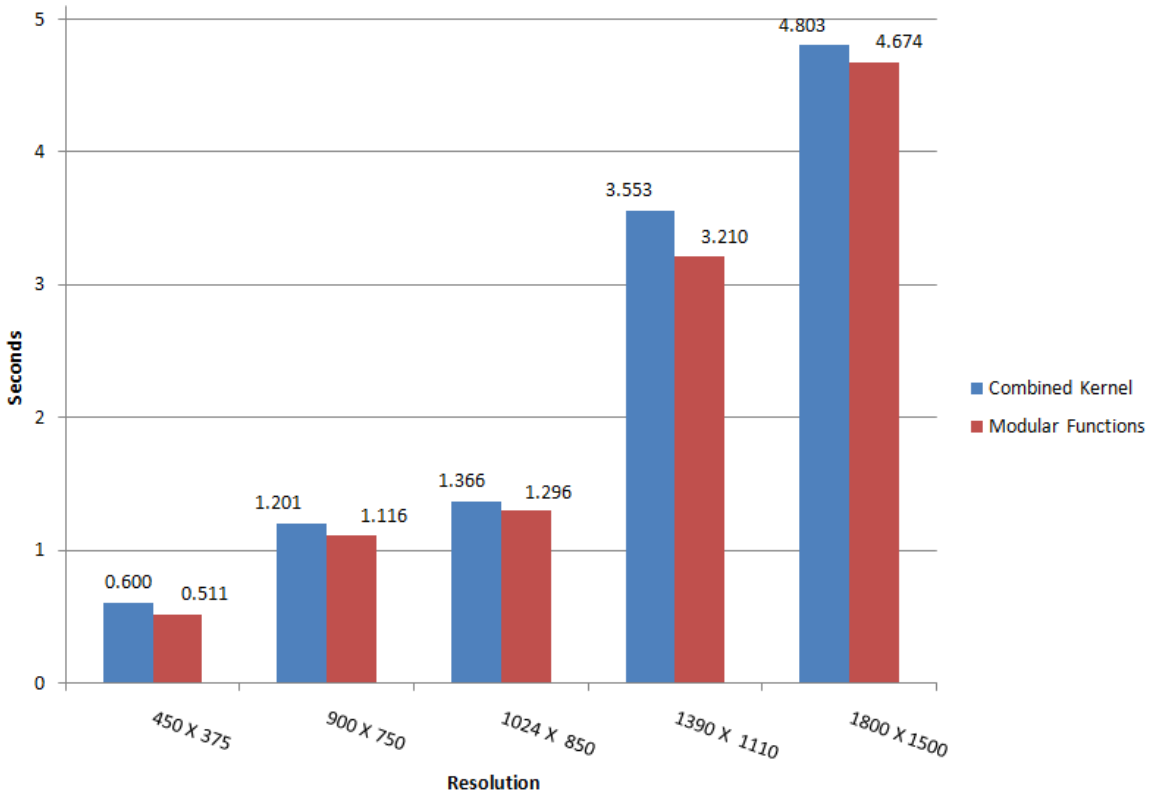


Figure 5.8: Modular framework vs Combined compute kernel

Although the EIB supports up to 30GB/s of memory bandwidth, this is not only from main memory, and it is possible that large amounts of data transfers will slow down the calculation of the final disparity map.

In order to test if the increased memory transfers would effect the overall speed of the calculation, a single compute kernel was developed which implements all stages of the stereo algorithm in a single compute kernel. The kernel uses the column based segmentation method for the entire process because it is needed to compute the aggregation steps.

The compute kernel takes as input a row of pixels the width of the column being executed, plus the number of disparity layers to be evaluated, from the target and reference images.

Both the modular framework and combined compute kernel are compared by recording the time to execute the adaptive weight algorithm at a number of different resolutions.

The results of the test can be seen in figure 5.8 and we can draw several conclusions from the results:

- The modular framework is seen to execute quicker for all resolutions tested. This is an unexpected result because the compute kernel removes the need to load data to and from the local store of the SPU, and also reduces the data transfers needed to load new object code to the SPU and associated overhead in executing the new task.

- The reason for the longer execution time of the combined compute kernel can be explained by examining the method of segmenting the image. Similar to the previous section, the reason for the slowdown can be attributed to the padding on the borders of the columns required by the aggregation stage.

- Instead of simply requiring additional data transfers, as is the case in the modular framework, the combined kernel needs to calculate the value in these overlapping regions. With a column width of 64 pixels it is necessary to calculate 32 extra pixels of initial disparity cost per column.

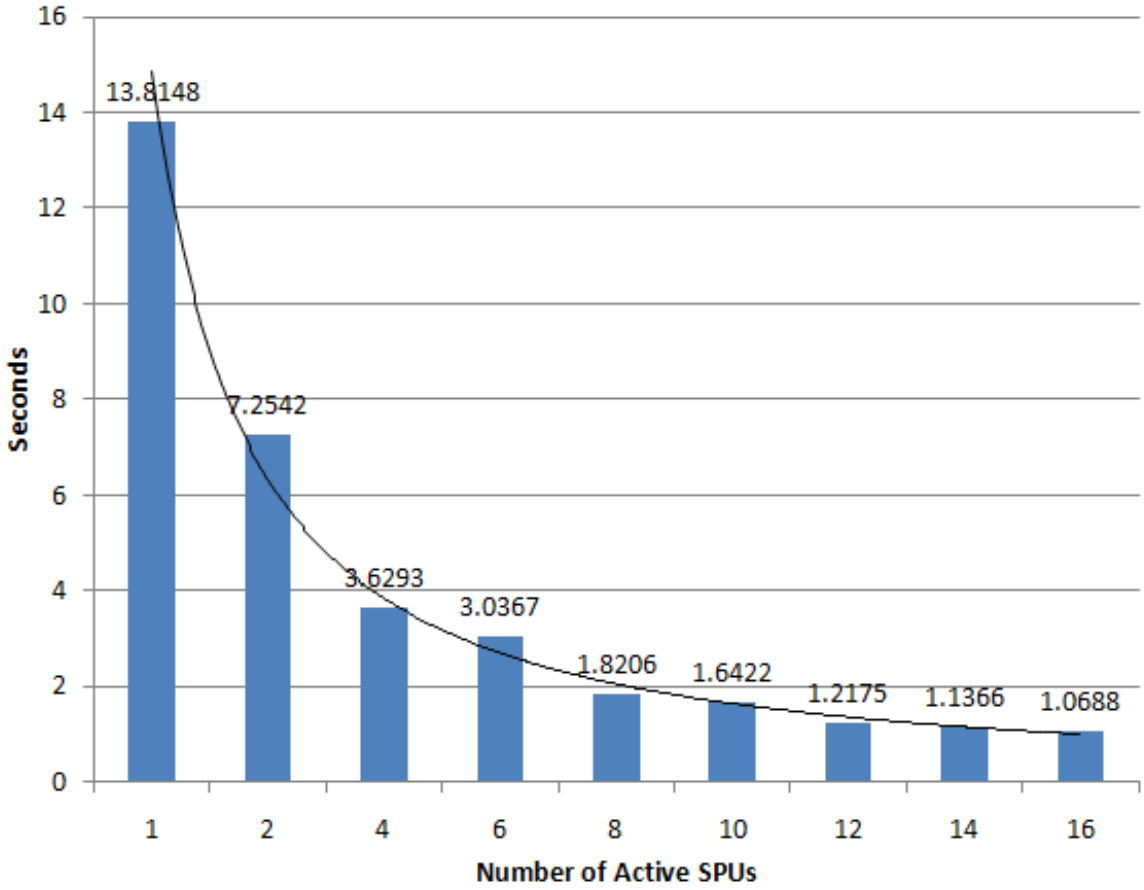


Figure 5.9: Effect of increasing number of SPUs

- The modular framework pre-calculates the initial disparity cost and transfers the data required for each column as it is needed, so extra calculations are avoided. This result is interesting because it highlights the difference between the time needed for data transfers verses increased computing cost.

5.5 Scaling over SPUs

The local stereo method was chosen because it is well suited for parallel execution. In order to evaluate how well the algorithm can be executed on a parallel architecture the following tests will evaluate the performance of the algorithm as it is segmented to take advantage of all available SPUs.

The test performs the same square window aggregation technique on a large image with a resolution of 1800X1500. The speed of the calculation is measured with the algorithm being executed using between 1 and 16 SPUs, which highlights how successfully the algorithm can be executed in parallel, and how well it scales to take advantage of available parallel resources.

Figure 5.9 plots the results of this test and the following conclusions can be made from the results:

- The time required to produce the disparity map reduces in a near linear fashion as the number of SPUs increases. This suggests that the algorithm is very well suited for parallel execution and there are limited dependencies which restrict the parallel nature of the application.
- This shows that as resources are made available the application is able to take full advantage of the extra processing power, and this gain in performance does not diminish as the execution becomes increasingly parallel.
- The excellent scalability of the local stereo algorithm means that if a shorter execution time is required it is only necessary to add extra processing power to directly effect the execution time.
- The fact that the application continues to scale near linearly as the number of SPUs increases from 8, available on a single cell processor, to 16, spread across two cell processors, suggests that the algorithm will be able to scale further as more cell processors are added.

5.6 Cell broadband verses x86

The final evaluation is designed to examine the speed improvements possible using the cell processor compared with a traditional x86 processor. The evaluation compares to adaptive weight algorithm as it is applied to an image with a resolution of 450X375 with a window size of 15X15.

Figure 5.10 shows the speeds obtained by running the algorithm on a single SPU, 16 SPUs, an X86 intel T9300 @ 2.50GHz and only running on the PPU. It should

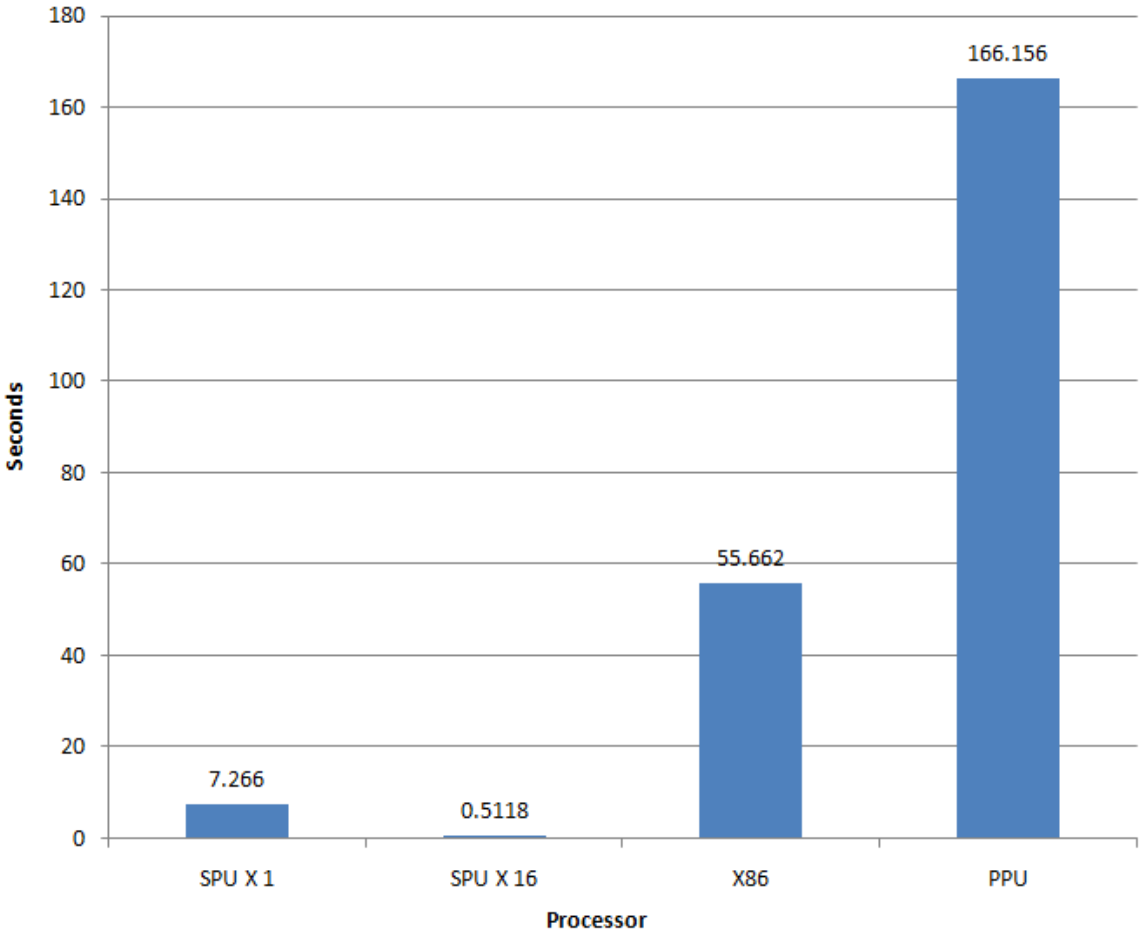


Figure 5.10: Comparison of different processor speeds

be noted that the implementation targeted for the cell processor has received more optimization, however, each of the implementations tested are full release builds with compiler optimization level O5. This includes auto SIMDization for the X86 platform.

The results clearly show how well suited the cell processor is for this type of application. A single SPU is able to complete the calculation in just 7.2 seconds, which is considerably faster than the X86 version which takes 55.66 seconds. When using all 16 SPUs, the cell implementation is two orders of magnitude quicker than the X86 version, completing in just 0.511 seconds.

The final comparisons highlight the difference between the PPU and SPU components of the cell processor. As described above, the PPU is designed as a general purpose device which manages the tasks while the SPU is targeted at heavy computation. The results highlight the difference in computing power with the PPU taking 166.15 seconds to compute the disparity map, considerably longer than either a single SPU or the X86 processor.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This thesis has described the implementation of a stereo algorithm on the cell processor.

The state of the art chapter of this thesis examined the current state of the art techniques for stereo vision. Global and local stereo algorithms were compared and analyzed for implementation on a parallel architecture. The design chapter focused on the design of the project framework. The square window and adaptive weight algorithms were introduced and examined. Also discussed here were the rationalisations of specific design choices. In the implementation chapter, the process of developing the project from the design outline to the complete and final implementation was discussed in detail, and specific issues that arose during development were examined. Finally, the evaluation chapter focused on several different aspects of the implementation to provide the results of a thorough analysis of the project.

We will now discuss the results of this project based on the original goals stated in the design chapter

Algorithm Evaluation & Selection

Evaluate the current state of the art in stereo methods and choose an efficient algorithm to implement and test:

A thorough analysis was conducted for the current state of the art algorithms, taking into account both global and local methods. During this analysis all factors contributing to the benefits and disadvantages of each algorithm were considered.

Based on this evaluation it was determined that global methods were unsuitable for implementation on the cell platform due to several factors, including their computational cost and inability to parallelize the entire algorithm.

Due to the results of this analysis, a local algorithm was deemed most suitable to achieve the goals of this project. Test results in the evaluation chapter support the choice of a local algorithm because it has been shown to scale well on the parallel platform and performs significantly faster than the sequential implementation of the same algorithm.

Modular Framework

Implement a modular framework to efficiently create disparity maps based on local stereo algorithms:

The modular framework was successfully implemented on the cell platform, with the exception of the disparity computation stage. During implementation it was discovered that the time taken to load the data onto the SPU for this simple task was excessive, and as a result this stage was combined with the aggregation stage to improve performance.

The choice of a modular framework is supported by the evaluation section where the modular framework was compared with a single computer kernel. The modular framework was shown to be faster than the single kernel and also provides a solid foundation for future work on stereo vision algorithms and general image processing on the cell platform.

Parallelization & Optimization

Exploit the power of the cell processor by parallelizing current techniques and optimizing for SIMD processing:

The evaluation shows that the implementation exploits the parallel nature of the cell processor well, but a number of issues were identified during evaluation that highlight

room for improvement. Specifically, the speed verses resolution tests indicate that the column segmentation of the image has a negative effect on efficiently harnessing the power of all SPUs.

Quality

Achieve high quality disparity maps:

The evaluation of the quality of the disparity maps produced shows that the adaptive weight algorithm does not produce as high a quality disparity map as the original implementation. This was expected as a number of simplifications were applied to increase the speed of the algorithm.

When compared with the disparity map results obtained by the openCV algorithm, both the square window and adaptive weight algorithm perform considerably better.

Real Time

Produce an algorithm that is capable of running in real time:

Although significant effort was spent optimizing the results the algorithms implemented are only capable of executing in real time for images a low resolution. The speed of the algorithm was shown to scale linearly based on the number of disparity levels to be evaluated. When only 16 disparity levels are evaluated it is possible to achieve real time performance with the square window algorithm with a window size up to 15X15.

The adaptive weight algorithm was found to be completely unsuitable for real time. This is because the algorithm does not perform well for small window sizes and has a high computational complexity.

6.2 Future Work

Generalize framework

The modular framework developed provides a basic framework for stereo algorithms. The framework provides the basic tools needs to load the images into memory efficiently and provides two methods of image segmentation for transferring data to the SPU. This forms an excellent basis for expansion to a general image processing platform.

Optimizations to support real time with more disparity levels

A major restriction of the implementation is the limited number of disparity levels which it is possible to evaluate in real time. Further optimizations to the code would benefit this with the possibility of redesigning the way the disparity space is stored to increase the speed at which it can be moved and stored on the SPUs.

Another optimization that can be applied to the square window method is instead of summing all the values within the window for each pixel, a box filter method which stores the cost for a boxes and only updates a single row for each iteration could be implemented. This would provide an increase in performance for real time application.

Calculate Column Width Automatically

The column width being fixed can have a large effect on the speed of the algorithm depending on the width of the image it is being applied to. To automate the process of selecting the appropriate column width so that it can be changed at runtime will remove the overhead discussed in the evaluation. As discussed there are many factors which need to be considered for deciding the column width so this task is non trivial.

Bibliography

[1] O. C. Team, "Opencv cell project." http://cell.fixstars.com/opencv/index.php/Status.

[2] M. Sonka, V. Hlavac, and R. Boyle, *Image Processing: Analysis and Machine Vision*. Thomson-Engineering, September 1998.

[3] R. Yang and M. Pollefeys, "A versatile stereo implementation on commodity graphics hardware," *Real-Time Imaging*, vol. 11, no. 1, pp. 7–18, 2005.

[4] D. Scharstein and R. Szeliski, "A taxonomy and evaluation of dense two-frame stereo correspondence algorithms," *International Journal of Computer Vision & Microsoft Research Technical Report MSR-TR-2001-81*, vol. 47, pp. 7–42, 2002.

[5] H. Hirschmuller and D. Scharstein, "Evaluation of cost functions for stereo matching," in *Computer Vision and Pattern Recognition, 2007. CVPR '07. IEEE Conference on*, pp. 1–8, June 2007.

[6] D. Scharstein, "Middlebury stereo evaluation," 2002. [Online; accessed 11-July-2009].

[7] K.-J. Yoon and I. S. Kweon, "Adaptive support-weight approach for correspondence search," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 28, pp. 650–656, April 2006.

[8] M. Gong, R. Yang, L. Wang, and M. Gong, "A performance study on different cost aggregation approaches used in real-time stereo matching," *International Journal of Computer Vision*, vol. 75, no. 2, pp. 283–296, 2007.

[9] T. Kanade and M. Okutomi, "A stereo matching algorithm with an adaptive window: theory and experiment," in *Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on*, pp. 1088–1095 vol.2, Apr 1991.

[10] M. Gong and R. Yang, "Image-gradient-guided real-time stereo on graphics hardware," in *3-D Digital Imaging and Modeling, 2005. 3DIM 2005. Fifth International Conference on*, pp. 548–555, June 2005.

[11] Y. Xu, D. Wang, T. Feng, and H.-Y. Shum, "Stereo computation using radial adaptive windows," in *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, vol. 3, pp. 595–598 vol.3, 2002.

[12] Q. Yang, L. Wang, and R. Yang, "Real-time global stereo matching using hierarchical belief propagation," in *BMVC06*, p. III:989, 2006.

[13] A. Klaus, M. Sormann, and K. Karner, "Segment-based stereo matching using belief propagation and a self-adapting dissimilarity measure," in *Pattern Recognition, 2006. ICPR 2006. 18th International Conference on*, vol. 3, pp. 15–18, 0-0 2006.

[14] Z.-F. Wang and Z.-G. Zheng, "A region based stereo matching algorithm using cooperative optimization," in *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pp. 1–8, June 2008.

[15] Q. Yang, L. Wang, R. Yang, H. Stewenius, and D. Nister, "Stereo matching with color-weighted correlation, hierarchical belief propagation, and occlusion handling," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 31, pp. 492–504, March 2009.

[16] L. Xu and J. Jia, "Stereo matching: An outlier confidence approach," in *European Conference on Computer Vision (ECCV)*, pp. 76–76, June 2008.

[17] D. Comaniciu and P. Meer, "Mean shift: a robust approach toward feature space analysis," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 24, pp. 603–619, May 2002.

[18] M. A. Fischler and R. C. Bolles, "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography," *Commun. ACM*, vol. 24, pp. 381–395, June 1981.

[19] H. Sugano and R. Miyamoto, "Opencv implementation optimized for a cell broadband engine processor," in *Digital Signal Processing Workshop and 5th IEEE Signal Processing Education Workshop, 2009. DSP/SPE 2009. IEEE 13th*, pp. 182–187, Jan. 2009.

[20] S. Birchfield and C. Tomasi, "Depth discontinuities by pixel-to-pixel stereo," in *Computer Vision, 1998. Sixth International Conference on*, pp. 1073–1080, Jan 1998.

[21] A. L. Beberg, D. L. Ensign, G. Jayachandran, S. Khaliq, and V. S. Pande, "Folding@home: Lessons from eight years of volunteer distributed computing," in *8th IEEE International Workshop on High Performance Computational Biology (HiCOMB 2009) in conjunction with the IEEE International Parallel and Distributed Processing Symposium (IPDPS 2009)*.

[22] IBM, "Ibm cell sdk," 2009. [Online; accessed 5-June-2009].

[23] IBM, *Accelerated Library Framework programmer's guide and API reference*, 2009 (accessed July 5, 2009). http://publib.boulder.ibm.com/infocenter/systems//topic/eiccn/eiccnkickoff.html?tocNode=toc:front/front.cmb/2/2/6/1/15/2/0/.

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

[25] M. Scarpino, *Programming the Cell Processor: For Games, Graphics, and Computation*. Prentice Hall, 2008.

[26] V. Srinivasan, A. K. Santhanam, and M. Srinivasan, *Cell Broadband Engine processor DMA engines, Part 2*, 2009 (accessed Sep 5, 2009). http://www.ibm.com/developerworks/library/pa-celldmas2/.

[27] IBM, *Cell Broadband Engine programming overview*, 2009 (accessed July 14, 2009). http://publib.boulder.ibm.com/infocenter/systems//topic/eiccb/eiccbprogrammingoverview.html?tocNode=toc:front/front.cmb/2/2/5/2/.

[28] IBM, *Cell Broadband Engine solution*, 2009 (accessed Aug 14, 2009). http://publib.boulder.ibm.com/infocenter/systems//index.jsp?topic=/eiccb/eiccbprogrammingoverview.html&tocNode=toc:front/front.cmb/2/2/5/2/.

[29] P. Sung, *SIMD programming on Cel*. MIT Opencourseware, 2009 (accessed June 2, 2009). http://ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-189January–IAP–2007/Recitations/detail/embed05.htm.

[30] D. A. Brokenshire, *Maximizing the power of the Cell Broadband Engine processor: 25 tips to optimal application performance*. IBM, 2009 (accessed July 18, 2009). http://www.ibm.com/developerworks/power/library/pa-celltips1/.