

Real Time Disparity Map Estimation on the Cell Processor

S. Cavanagh¹ and M. Manzke²

¹MSc Interactive Entertainment Technology, Trinity College Dublin

²Graphics Vision and Visualisation GV2 group, Trinity College Dublin

Abstract

Stereo vision attempts to regain depth information from a pair of 2D images. This information can be used in a wide range of areas including robotics, augmented reality, 3D TV and movie post production. This paper will describe the development of a stereo algorithm on the cell processor. The implementation will be evaluated for both quality and speed, demonstrating the effectiveness of the cell processor as an image processing platform. The restrictions imposed by the platform and issues which are faced when developing computer vision applications on the cell will be discussed and the methods taken to implement the algorithm will be described.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware architecture—Parallel processing I.4.8 [Image Processing and Computer Vision]: Scene Analysis—Stereo

1. Introduction

The field of stereo computer vision has become an active area of research over the past number of years. In the past, the lack of sufficient processing power was a major limiting factor in image processing. With the advent of parallel technologies such as the cell processor and GPU, new avenues and possibilities have opened for real time applications of computer vision.

This paper will describe the implementation of a stereo algorithm on the cell processor. The evaluation of the implementation demonstrates the effectiveness of the cell processor as an image processing platform. The restrictions imposed by the platform will be discussed and the methods taken to implement the algorithm on a parallel architecture will be described.

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 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 certain computationally expensive operations in significantly less time than using other contemporary processors. Parallel processing 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 in parallel.

2. State of the Art

The Middlebury University [Sch02] provides a list of the current top performing stereo algorithms ranked by the quality of the disparity map produced.

Klaus et al [KSK06] present a stereo algorithm that achieves the top rank on the Middlebury dataset with an average of only 4% incorrect pixels. This method uses segment based belief propagation which takes a significant amount of processing power. The average runtime of this algorithm is between 14 to 25 seconds.

Yang et al [YWY*09] describe the implementation of a hierarchical BP algorithm which achieves a similar quality with a runtime of approximately one minute on serial hardware. The authors suggest that aside from the mean shift segmentation, this algorithm would be well suited for parallel execution on GPU or cell hardware.

Yang et al [YEA08] achieve near real time performance by utilizing the parallel processing of the GPU while maintaining an average of 5.8% incorrect pixels. Wang et al

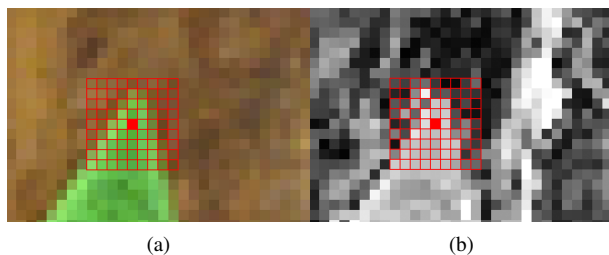


Figure 1: Square Window Aggregation

[WLG*06] evaluate both a CPU and GPU version of their dynamic programming stereo framework, achieving a speed up of over two orders of magnitude on the GPU hardware while maintaining a low average of 9.8% incorrect pixels.

Finally Gong et al [GYWG07] provide an overview of a number of local stereo algorithms that operate in real time on GPU hardware.

The OpenCV library is an open source image processing library aimed at real time applications. The OpenCV on Cell project is an implementation of sections of this library optimized for the cell processor. Sugano and Miyamoto [SM09] show that the use of the cell processor significantly increases the performance of many functions provided by the library.

3. Stereo Vision

Stereo algorithms vary in their implementation, however it is observed that most stereo algorithms perform (subsets of) the following four steps [SS02]:

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

Initial Matching Cost Computation The purpose of this stage is to calculate a simple estimation of the matching cost for each pixel in the image at every disparity level being evaluated. This is achieved by comparing the stereo pair on a pixel by pixel basis at each disparity. The end result of this computation is a 3D matrix of costs, known as the disparity space volume, which contains the matching cost for each pixel at each disparity level.

Scharstein and Szeliski [SS02] state in their evaluation of stereo algorithms that the most common dissimilarity measures are Squared intensity Difference (SD) and Absolute intensity Difference (AD) which offer little difference in performance.

The intensity difference for colour pixels is calculated by first calculating the intensity difference for each of the red, green and blue channels. These are then com-

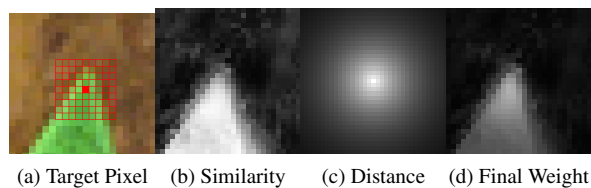


Figure 2: Adaptive Weight Window Calculation

puted using weighted values to determine the final cost.

$$\text{cost} = (R * 0.299 + G * 0.587 + B * 0.114)$$

A number of variations exist for these basic matching costs including Truncated and Scaled Absolute intensity Difference (TSAD) [YK06]. 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. For a complete overview of the advantages and disadvantages of most matching costs see [SS02].

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 signal to noise ratio is reduced by averaging the cost for each pixel over a support window surrounding the target pixel.

Window based aggregation algorithms differ in the way that the window is chosen and also how they calculate the cost based on this window.

Two separate aggregation methods were implemented for this paper which provides a comparison for both quality and speed.

- **Square Window** 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 level. Figure 1 shows a zoomed in section of an image. This is useful to illustrate the use of a square window. The area of the reference image the window encloses can be seen in (a), while (b) shows the associated costs for a single disparity level. The target pixel will have its cost updated with the sum of the cost of all pixels within the window.

- **Adaptive Weight** The second method focuses primarily on quality over the speed of calculation and is based on the Adaptive Weight algorithm [YK06].

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 relevant that cost is to the pixel under consideration.

Figure 2 shows the process of developing the pixel weights. (a) shows the target pixel, (b) shows the intensity values which represent the similarity weight, (c) shows the proximity weight, and (d) shows the combined weight for each pixel.

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 [YK06] :

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

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

Two modifications were applied to both the x86 and cell implementations of the original algorithm to improve speed and reduce memory bandwidth. First, the adaptive window weights will be calculated only once based on the reference image. This simplification significantly reduces the computation required for each disparity. 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 and also reduce the size required for the disparity space volume which is transferred to and from the SPU memory.

Disparity computation The task of the disparity computation stage is to process the data stored in the 3D disparity space 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. 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 because there is no smoothness constraint on the disparity map created, but instead it is calculated on a pixel by pixel basis. This per pixel approach allows errors which arise from incorrectly matched pixels. In order to remove these spurious pixels, it is possible to implement a median filter which is efficient at removing small patches of bad pixels while maintaining sharp edges.

4. Cell Implementation

4.1. Accelerated Library Framework

The Accelerated Library Framework (ALF) provides a programming environment for data and task parallel applications. The API provides a set of interfaces to simplify development on multi-core systems [IBM09c].

4.1.1. ALF Structure

ALF applications must follow a basic structure in order to properly use the ALF runtime. There are two main structures defined to describe jobs that are executed by the ALF runtime on the SPUs.

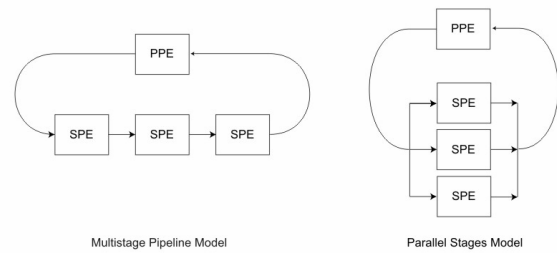


Figure 3: Parallel Implementation Models

- A **Task** is an abstract definition of a processing job [Sca08]. It contains specific information about the task to be executed, for example, the data to be transferred, how many SPUs to use, and what compute kernel to execute.
- A **Workblock** is a concrete invocation of a task that will be executed on the SPU.

4.1.2. Memory Management 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 it into six main buffers.

- The **Task Context Buffer** contains common data across multiple workblocks in a single task.
- The **Parameter Buffer** is passed to a workblock at the time of its execution and contains specific information about the workblock being executed.
- The **Input Buffer** contains the data which has been transferred to the block to be processed.
- The **Output Buffer** is where all output data from the compute kernel back to main memory must be saved.
- The **Overlapped IO Buffer** is useful when you can overwrite input data with output data to maximize the memory available on the accelerator (SPU).
- The **Stack Buffer** contains variables created and used during the computation kernel.

4.2. Parallel Algorithm Pipeline

The process of segmenting the algorithm into blocks which can be processed using the parallel architecture of the cell can be approached in a number of ways. Figure 3 shows an example of the two main methods of parallingizing the tasks:

- A **Multi Stage Pipeline** is created by assigning a specific job in the pipeline to each SPU. The SPU will 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

Array of structure																
R1	G1	B1	R2	G2	B2	R3	G3	B3	R4	G4	B4	R5	G5	B5	PAD	
R6	G6	B6	R7	G7	B7	R8	G8	B8	R9	G9	B9	R10	G10	B10	PAD	
R11	G11	B11	R12	G12	B12	R13	G13	B13	R14	G14	B14	R15	G15	B15	PAD	
R16	G16	B16	R17	G17	B17	R18	G18	B18	R19	G19	B19	R20	G20	B20	PAD	
Structure of Arrays																
R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	
R17	R18	R19	R20	R21	R22	R23	R24	R25	R26	R27	R28	R29	R30	R31	R32	
G1	G2	G3	G4	G5	G6	G7	G8	G9	G10	G11	G12	G13	G14	G15	G16	
G17	G18	G19	G20	G21	G22	G23	G24	G25	G26	G27	G28	G29	G30	G31	G32	
B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12	B13	B14	B15	B16	
B17	B18	B19	B20	B21	B22	B23	B24	B25	B26	B27	B28	B29	B30	B31	B32	

Figure 4: Data Layout in Memory

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. 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.

”Multi-stage pipelining is typically avoided because of the difficulty of load balancing. In addition, the multi-stage model increases the data-movement requirement” [IBM09b]. Image processing algorithms with limited dependencies between pixels can be successfully segmented into tasks that can be executed as task parallel blocks.

4.3. Memory Management & Image Segmentation

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 [Sca08].

4.3.1. 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 [IBM09a].

Data is stored in the SPU LS as 16 byte aligned blocks which can be accessed as vectors. 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. For variables that do not use the entire 16 bytes it is necessary to create some padding data to fill 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. 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

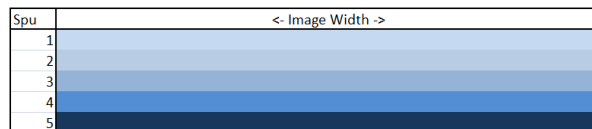


Figure 5: Cyclic Distribution: Each SPU operates on a single line of the image at a time

a Structure of Arrays (SOA) which stores the image data in memory as separate continuous arrays for each of the channels. Figure 4 illustrates how these would appear in memory.

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. By using the SOA method for storing data we can pack 16 bytes of a single channel into the block of memory.

AOS data-packing often produces smaller code sizes, but it typically executes poorly and requires significant loop-unrolling to improve its efficiency [AAM*08].

Sung [Sun09] compares the development of a 3D simulator using euler integration implemented with an AOS and SOA structure. The AOS structure takes 300ms to complete and the SOA structure takes only 80ms. 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.

4.3.2. Image Data Segmentation

The task of segmenting the image data to most efficiently use the SPUs depends on the requirements of the compute 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 5 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.

Real Time Disparity Map Estimation on the Cell Processor



Figure 6: Block Distribution: Each SPU operates on a single column of data

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 which is 1920 pixel wide, 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 aggregation 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 6 shows an image segmented into five columns which are processed on five SPUs.

There is a drawback to segmenting the images onto separate 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 transferred back to main memory. The need to load overlapping data adds an overhead that cannot be avoided. The actual cost per row for this method is calculated using:

```
rowSize = (usableBlocks + adjacent-
Blocks) * (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 overlapping data, but it is unavoidable due to the limited memory on the SPU, the requirements of the algorithm, and the use of 16 pixel wide blocks to store the RGB channels.

The workblocks 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 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 workblocks that execute on a column of data.

Consider the example of aggregating the cost of a single pixel with a window size of 15X15. The number of pixels required to produce a result is 225 pixels. With 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, transferring 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 transferred to the SPUs.

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 on the SPU. 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 * (num-
Rows + windowSize)
```

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 significantly reduces the amount of data which must be transferred, and allows the solution to scale successfully to much larger images by removing the link between image width and memory usage on the SPU. This method makes the memory usage completely predictable based on the number of disparity levels and window size.

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.

4.4. Optimizations

In order to exploit the power of the cell processor it is necessary to optimize the code it will process. Although 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.

- **SIMD Intrinsics** The use of SIMD instruction is essential to take advantage of the SPE processing speed. The cell libraries provide the SIMD intrinsics which allow the programmer to directly implement SIMD instructions
- **Branch Prediction** Incorrectly predicted branches cause an 18 cycle stall on the SPU [Bro09]. When branches cannot be avoided and the more likely path to be taken is known at compile time, it is necessary to direct the compiler to prioritize the correct path.
- **Loop Unrolling** Loops are a key requirement of many applications, however they introduce extra branch code

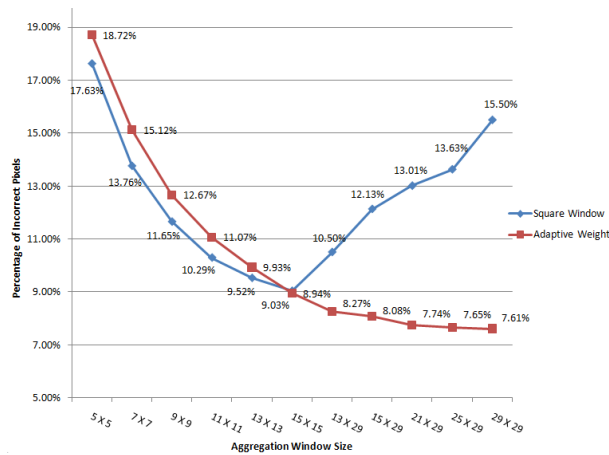


Figure 7: Disparity map quality

which does not perform well on the SPU. Loop unrolling simply involves flattening out the loop by manually copying the code multiple times. It is possible to completely unroll loops when there is a constant number of iterations.

5. Evaluation & Results

This section of the paper evaluates the performance of the implementation with respect to the quality of the disparity maps produced, on the performance of the cell processor vs an x86 implementation, and also on how well the algorithm performs on a parallel architecture.

5.1. Aggregation Techniques

The Middlebury stereo dataset [Sch02], which includes a ground truth disparity map for each stereo image set, was used as a means to evaluate the implementation. Disparity maps are evaluated based on the number of incorrect pixels in the image. Pixels with disparities that differ by more than 1.0 from the ground truth are deemed to be incorrect [HS07]. Figure 7 illustrates the results obtained for support window sizes ranging from 5X5 to 29X29. 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.
- The square window algorithm has a peak performance with a window size of 15X15 after which the quality of the disparity map begins to 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



(a) Cones Image (b) Cones Disparity Map

Figure 8: Middlebury Dataset Images

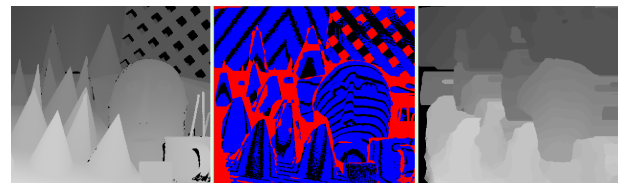


Figure 9: Comparison with ground truth(left), difference(center) and 29X29 square window(right)

the borders of objects. Figure 9 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.

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

5.2. Speed versus Resolution

The next stage of the evaluation will test the effect of different resolution images on the running time of the implementation. The test is conducted by running the adaptive weight algorithm on image sets with resolutions ranging from 450X375 to 1800X1500. The results of the test can be

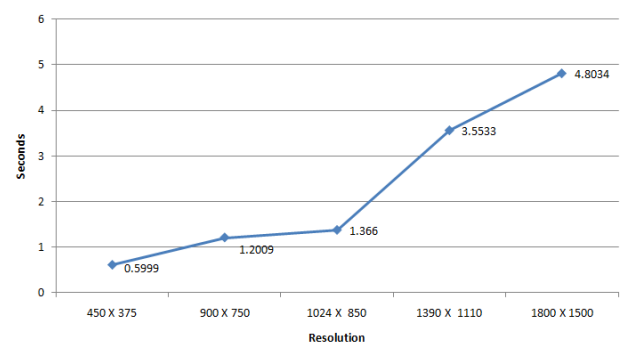


Figure 10: Computation time for different resolutions

Real Time Disparity Map Estimation on the Cell Processor

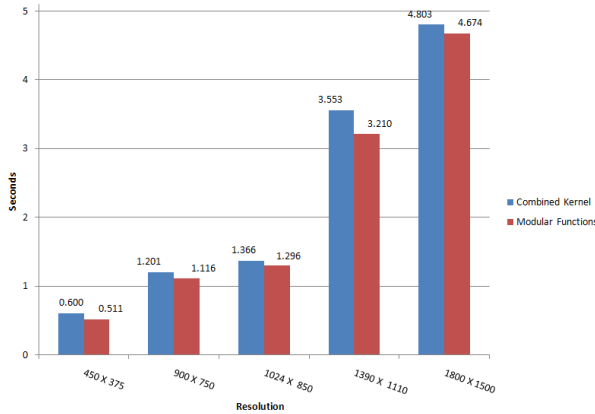


Figure 11: Modular framework vs Combined kernel

seen in Figure 10. Irregularities can be observed which are caused by the segmentation of the image into columns.

- 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. 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.
- Second, there is a large increase in processing time between the 1024X850 image and the 1390X1110 image. This is caused because the width of the image is large enough to make use of all of the SPUs $1024 / 64 = 16$ SPUs. The number of SPUs required for the second image is - $1390 / 64 = 21$. This means five SPUs will have to process two columns of the image, significantly increasing processing time.

5.3. 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.

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. The kernel uses the column based segmentation method for the entire process because it is necessary for the aggregation stage. The results of the test can be seen in figure 11 and we can draw several conclusions:

- 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.

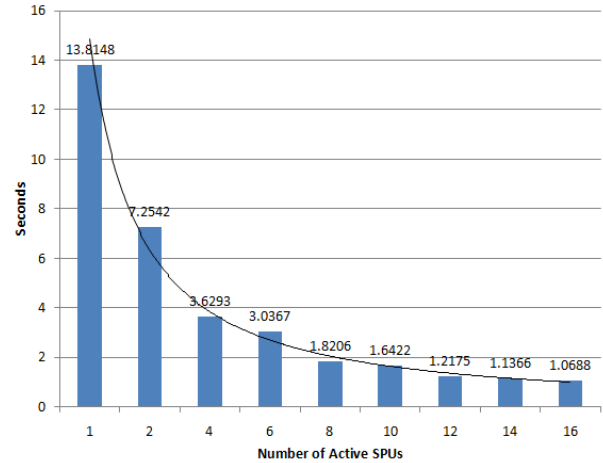


Figure 12: Effect of increasing number of SPUs

- 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.
- 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.

5.4. Scaling over SPUs

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 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. This highlights how successfully the algorithm can be executed in parallel, and how well it scales to take advantage of available parallel resources. Figure 12 plots the results of this test and shows:

- The time required to produce a disparity map reduces in near linear fashion as the number of SPUs increases, suggesting the algorithm is well suited for parallel execution.
- 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.

5.5. Cell broadband versus x86

The final evaluation is designed to examine the speed improvements possible using the cell processor compared with an x86 processor. The evaluation compares the adaptive

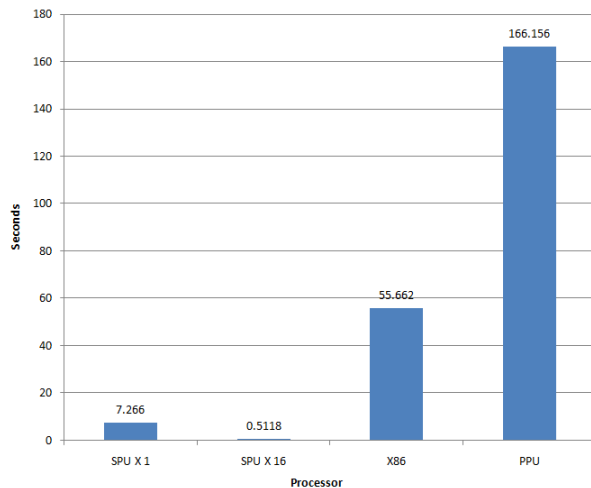


Figure 13: Comparison of different processor speeds

weight algorithm as it is applied to an image with a resolution of 450X375 and window size of 15X15.

Figure 13 shows the speeds obtained by the algorithm on only the PPU, a single SPU, 16 SPUs, and a x86 intel T9300 @ 2.50GHz. It should be noted that the implementation targeting the cell processor has received more optimization.

The results 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, 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, taking just 0.511 seconds.

The final comparisons highlight the difference between the PPU and SPU components of the cell processor. The PPU is designed as a general purpose device which manages the tasks while the SPU is targeted at heavy computation.

6. Conclusion

In this paper we have described the implementation of a stereo vision algorithm on the cell platform. The implementation was evaluated, demonstrating the effectiveness of the cell processor as an image processing platform.

The results show that a significant performance increase can be obtained, however, there are a number of restrictions imposed by the platform and most algorithms will require a significant amount of modification to benefit from implementation on the cell platform.

References

[AAM*08] ALMOND C., AREVALO A., MATINATA R. M., PANDIAN M. R., PERI E., RUBY K., THOMAS F.: *Programming the Cell Broadband Engine - Architecture: Examples and Best Practices*. IBM Redbooks, 2008.

[Bro09] BROKENSHERE D. A.: *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/>.

[GYWG07] GONG M., YANG R., WANG L., GONG M.: A performance study on different cost aggregation approaches used in real-time stereo matching. *International Journal of Computer Vision* 75, 2 (2007), 283–296.

[HS07] HIRSCHMULLER H., SCHARSTEIN D.: Evaluation of cost functions for stereo matching. In *Computer Vision and Pattern Recognition, 2007. CVPR '07. IEEE Conference on* (June 2007), pp. 1–8.

[IBM09a] 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/>.

[IBM09b] 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/>.

[IBM09c] IBM: *Accelerated Library Framework programmer's guide & 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/>.

[KSK06] KLAUS A., SORMANN M., KARNER K.: Segment-based stereo matching using belief propagation and a self-adapting dissimilarity measure. In *Pattern Recognition, 2006. ICPR 2006. 18th International Conference* (0-0 2006), vol. 3.

[Sca08] SCARPINO M.: *Programming the Cell Processor: For Games, Graphics, and Computation*. Prentice Hall, 2008.

[Sch02] SCHARSTEIN D.: Middlebury stereo evaluation, 2002. [Online; accessed 11-July-2009].

[SM09] SUGANO H., MIYAMOTO R.: 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* (Jan. 2009).

[SS02] SCHARSTEIN D., SZELISKI R.: A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International Journal of Computer Vision & Microsoft Research Technical Report MSR-TR-2001-81* 47 (2002), 7–42.

[Sun09] SUNG P.: *SIMD programming on Cell*. 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>.

[WLG*06] WANG L., LIAO M., GONG M., YANG R., NISTER D.: High-quality real-time stereo using adaptive cost aggregation and dynamic programming. In *3DPVT '06: Proceedings of the Third International Symposium on 3D Data Processing, Visualization, and Transmission (3DPVT'06)* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 798–805.

[YEA08] YANG Q., ENGELS C., AKBARZADEH A.: Near real-time stereo for weakly-textured scenes. pp. xx–yy.

[YK06] YOON K.-J., KWEON I. S.: Adaptive support-weight approach for correspondence search. *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 28, 4 (April 2006), 650–656.

[YWY*09] YANG Q., WANG L., YANG R., STEWENIUS H., NISTER D.: Stereo matching with color-weighted correlation, hierarchical belief propagation, and occlusion handling. *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 31, 3 (March 2009), 492–504.