3D-TV Rendering on a Multiprocessor System on a Chip

Xing Li[*,**], Jos van Eijndhoven[*], Ben Juurlink[**]

[*]Philips Research Laboratory,
Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands
[**]Computer Engineering Laboratory, Faculty of Electrical Engineering,
Mathematics, and Computer Science, Technische Universiteit Delft
Mekelweg 4, 2628 CD Delft, The Netherlands

X.Li-ET@its.tudelft.nl

Abstract – **In this paper we describe how two 3D-TV rendering algorithms have been mapped onto a chip multiprocessor named Wasabi. This platform contains several TriMedia processors that communicate via a shared memory, fast message-passing channels to support multi-chip systems, and some application-specific co-processors. By mapping 3D-TV rendering applications to Wasabi, the performance figures are obtained not only to check the feasibility of the algorithms and mappings, but also to match the application requirements with the hardware architecture. The results show that both algorithms scale with the number of processors. The first algorithm makes viewer see 2D effect without glasses. However, to get 3D effect, the viewer must wear a pair of special glasses. And it can be executed in real-time on a single TriMedia processor. The 3D effect produced by the second algorithm can be seen without wearing special glasses. To execute this algorithm in real-time, 16 TriMedias are needed.**

Key words - **3D-TV rendering; optimization; mapping; multiprocessor; system on a chip**

I. INTRODUCTION

Modern embedded systems have to support a wide range of applications. This implies that they have to be highly programmable. This can be achieved by applying programmable microprocessors, but in order to satisfy the performance requirements of different applications, several microprocessors have to be integrated on a single chip.

The drawback of existing Philips programmable platforms is that the programming paradigm is locked to Trimedia Streaming Software Architecture (TSSA). Wasabi solution [4] can overcome this drawback by providing an industry standard programming model that offers a single shared linear memory, complete hardware cache coherence, and high-speed synchronization primitives

However, there are not many targeted applications are available yet for this platform. In this paper we describe how two 3D-TV rendering algorithms [2] [3] have been mapped onto Wasabi, which not only checks the feasibility of the algorithms and

mappings, but also matches the application requirements with the hardware architecture.

In order to implement the algorithms, the following approach was taken. First, sequential C code was developed. Thereafter, we optimized the code by performing algorithmic and arithmetic optimizations (such as reducing the control flow, floating-point to fixed-point conversion, etc.) and vectorized using the special, SIMD-like media instructions provided by the TriMedia. Finally, multithreaded programs were developed.

The results show that algorithmic and arithmetic optimizations substantially improve the performance of both algorithms, in one case by more than a factor of 10. Vectorization also improves performance, but not as much as theoretically achievable because although 4 bytes can be packed in a single word we cannot exploit 4-way parallelism because intermediate results can be larger than one byte. The results also show that both multithreaded programs scale well with the number of processors.

This paper is organized as follows. In Section II, the preliminaries are given. Section III describes how the glasses-based algorithm [2] was implemented, optimized, and vectorized for Wasabi and also presents its experimental results. The implementation of the more elaborate algorithm [3] that does not require wearing special glasses and its performance are given in Section IV. Finally, in Section VI some conclusions are drawn and directions for future work are proposed.

II. PRELIMINARIES

Depth map and RGBD format

Philips promotes the so-called *image+depth* or RGBD format as the 3D video standard. As illustrated in Figure 1, it can be seen that a per-pixel depth map accompanies the normal image. Every pixel of the normal image is composed of 3 basic colors - red, green, and blue (RGB) – each of which is a value between 0 and 255. The values of R, G, and B from the normal image and the depth information from the depth map can be packed in a single 32-bit word, which allows SIMD operations to process it. The depth map is a grey level picture (R=G=B), in which the objects closer to the camera are brighter than the objects further from it.

Figure 1: Normal image and its depth map.

3D display categories and 3D effect

A categorization is given in Figure 2.

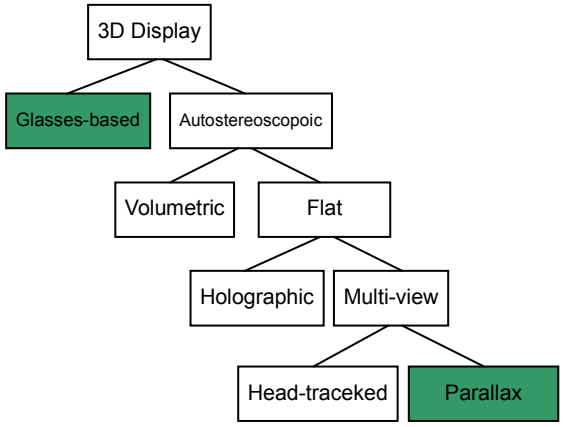

Figure 2: 3D display categories. (example taken from [2, page 1]).

The algorithm described in Section III is for glasses-based systems. The algorithm discussed in Section IV is on the aspect of parallax.

There are several kinds of parallax that can create a 3D effect. Philips Research advocates the so-called *9-view solution* (each view only has one basic color), in which people see a frame with 9 views (colors) interleaved through a special lens. As a result, people see the 3D effect.

III. MAPPING THE GLASSES-BASED ALGORITHM ONTO WASABI

Description of the algorithm

This algorithm is a combination of 3D and monoscopic viewing and associated with glasses-based display system. The viewer will see the 3D effect when he or she wears a pair of special glasses and a high-quality 2D video when he or she takes them off.

When the viewer wears a pair of special glasses, each of his eyes will receive a different image which will be referred to as I_L and I_R. I_L and I_R can be calculated from RGBD data by warping luminance (and color) according to [1]:

$$I_L(x,y) \approx I(x,y) - D(x,y)\frac{\partial}{\partial x}I(x,y) \quad (1)$$

$$I_R(x,y) \approx I(x,y) + D(x,y)\frac{\partial}{\partial x}I(x,y) \quad (2)$$

In these equations, D contains the depth information and I is the warping of luminance of the original image. Figure 3 illustrates Equation (1) and (2).

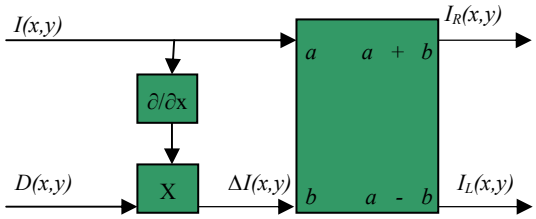

Figure 3: data serves as content for the 3D/mono system.

Without glasses, the monoscopic viewer will see I_L and I_R superimposed in both eyes. It is easy to derive from Equation (1) and (2) that:

$$I_{MONOSCOPIC} = \frac{I_L + I_R}{2} \quad (3)$$

Implementation

In the original implementation, D is calculated using the following formula:

$$D(x,y) = 2 \bullet (\frac{R(x,y)}{255.0} - 0.5) \quad (4)$$

where *R(x,y)* is the red value of the pixel with coordinates *(x,y)* in the depth map. Because the depth map is a grey level picture, *R(x,y)* can be replaced by *G(x,y)* or *B(x,y)*. From Equation (4) it follows that *D(x,y)* is in range *[-1,+1]* and that it can take on 256 different values. Furthermore, the range can be changed by multiplying D with another integer (the variable disparity-range in Figure 4).

Pseudo-code of the algorithm is depicted in Figure 4(a). The actual source code is written in C.

Optimizations

The algorithm was optimized for the TriMedia processor using the following approach:

a) First, algorithmic and arithmetic

optimizations are performed. Examples of such optimizations are reducing the control flow, converting floating point operations to fixed point (integer) ones, etc.

b) Second, vectorization is employed. The R, G, B, and D data are packed into 32-bit words, which allows to process them in parallel using the SIMD instructions of the TriMedia.

Figure 4 illustrates the algorithmic and arithmetic optimizations.

```
switch = left or right;
for each pixel
{if (the pixel is not on the edge)
      {D(x,y)=2.0*(R(x,y)/255.0-0.5);
      ΔI(x,y)=(I(x-1,y)-I(x+1,y))/2;
      ΔI(x,y)= ΔI(x,y)*D(x,y)*disparity-range;
      if (switch=right)
            Out_I(x,y)=I(x,y)+ ΔI(x,y);
      else
            Out_I(x,y)=I(x,y)+ ΔI(x,y);
      }
else
      Out_I(x,y)=I(x,y);
}                          (a)
```

```
for each pixel
{
condition=the pixel is on the edge;
D(x,y)=(R(x,y)-128)*disparity-range;
ΔI(x,y)=(I(x-1,y)-I(x+1,y))*D(x,y)>>8;

I_R(x,y)=I(x,y)+ΔI(x,y);
I_L(x,y)=I(x,y)- ΔI(x,y);

Out_I_R(x,y)=mux(condition,I(x,y),I_R(x,y));
Out_I_L(x,y)=mux(condition,I(x,y),I_L(x,y));
}                          (b)
```

Figure 4: Algorithmic and arithmetic optimizations.

In Figure 4(a), the inner control flow (if (switch=right)) is removed by rendering the left and right image simultaneously. The outer control flow (if (the pixel is not on the edge)) is eliminated by employing the *mux* operation, which returns its second operand if the condition evaluates to be true and its third operand otherwise. Furthermore, the underlined floating point operations are converted to fixed point (integer) ones which replaces the division operation by a shift. In Figure 4, every operation depicted in italics actually consists of three operations associated with R, G and B.

Figure 5 illustrates how the operations: $\Delta I(x,y)=(I(x-1,y)-I(x+1,y))*D(x,y)>>8)$ can be implemented using the SIMD instructions of the TriMedia. The operation above includes the following three operations in deed:
$\Delta R(x,y)=(R(x+1,y)-R(x-1,y))\cdot D>>8$,
$\Delta G(x,y)=(G(x+1,y)-G(x-1,y))\cdot D>>8$, and
$\Delta B(x,y)=(B(x+1,y)-B(x-1,y))\cdot D>>8$.
Without vectorization, those operations will be executed sequentially.

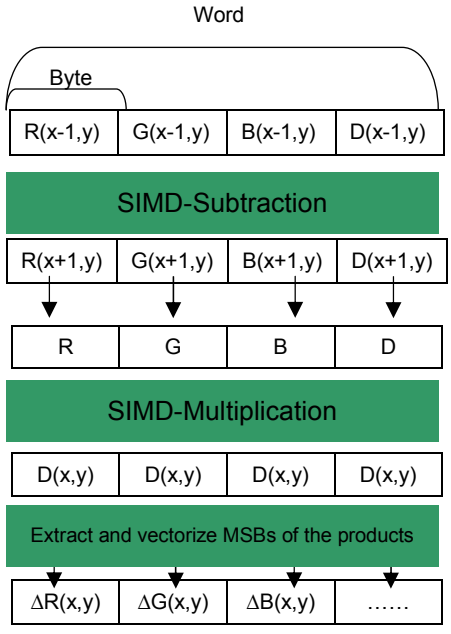

Figure 5: Employing SIMD instructions.

From Figure 5, it can be seen that the operation is decomposed into a sequence of two SIMD operations -- subtraction and multiplication. In the SIMD-Multiplication, the MSBs of the products are extracted and vectorized, which is equivalent to shift every product 8 bits to the right. The result in the least-significant byte is discarded. The other operations in italics in Figure 4(b) can be vectorized in the similar way.

Multi-threading

To benefit from multiple processors, the program must be parallelized using multiple threads. The scheduler of Wasabi will automatically balance the workload on each processor by assigning threads to processors that are idle. Since the pixels of the original image and the depth map are stored in large arrays, the idea of multi-threading is to divide the arrays into equal pieces and to have each thread process a different piece (segment of the screen).

.

Experimental results

The performance figures were obtained using a cycle-accurate simulator of the Wasabi architecture. All simulations associated with this algorithm were done using 2 frames of 720x576 input pixels.

Figure 6 depicts the number of cycles required by the original code and that after the optimizations and vectorization. It can be seen that algorithmic and arithmetic optimizations improve performance by a factor of 12.3x. Vectorization yields another factor of 2.1x, so 25.8x in total.

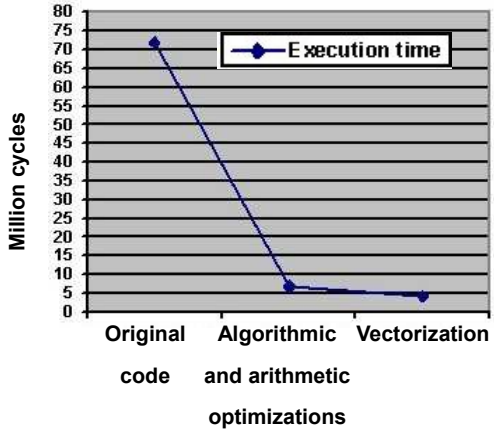

Figure 6: Execution time (the number of cycles) of the different program variants on a single TriMedia processor.

Figure 7 shows the number of cache stall cycles that each program variant incurs.

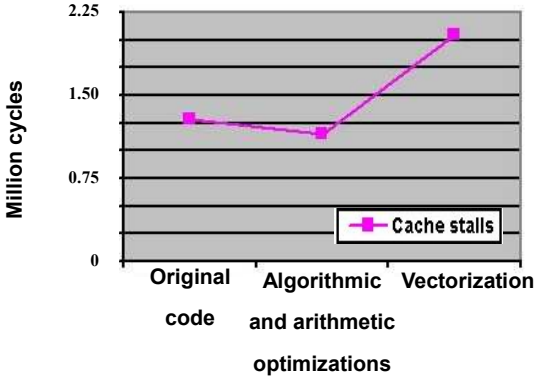

Figure 7: Number of cache stall cycles each program variant incurs.

In Figure 7, the numbers of cache stall cycles decreases after the algorithmic and arithmetic optimizations. The reason is that in the original code I_R and I_L are computed sequentially. However, the cache cannot store the whole input image and its depth map. Therefore, when the second image is calculated, some of the input data, which was used for calculating of the previous image, must be reloaded from memory. In the program variant after algorithmic and arithmetic optimizations, I_R and I_L are computed interleaved so that the input image data is reused. After vectorization, the

number of cache stall cycles increases. Because the algorithm is accelerated, the processor accesses data much more quickly, which creates some extra cache stalls.

The TriMedia processor can issue one very long instruction word (VLIW) every cycle and each VLIW consists of 5 *Basic_operations* (called *slots*) such as add, load, etc. In practice there are two causes for not achieving an actual launch rate of 5ops/cycle. (a) In compile time, the compiler might not be able to reveal sufficient ILP in the application program to always fill the 5 issue slots. In such cases it will insert useless 'NOP' operations. (b) In run time, as result of instruction cache or data cache misses, the processor will incur stall cycles. In stall cycles no new instructions are launched. Figure 8 shows the combined effect of these two causes in variable Efficiency. The Efficiency reflects the compactness of those *Basic_operations* in the VLIWs and can be define as:

$$\text{Efficiency} = \frac{Basic\_operations}{5 \bullet Execution\_time} \quad (5)$$

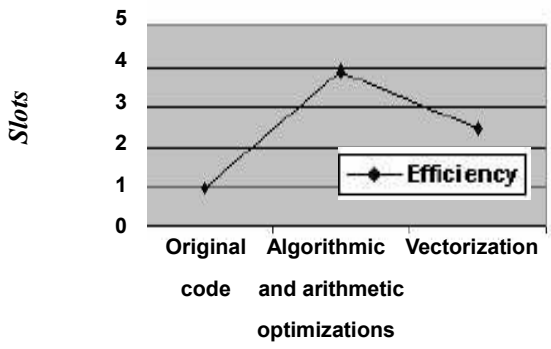

Figure 8: IPC for Instructions-per-cycle of the program variants.

In Figure 8, the Efficiency of the original code is poor because it contains many control and data dependences. The algorithmic and arithmetic optimizations eliminate those dependences and the corresponding program variant achieves high Efficiency (almost 80% corresponding to 4 of the 5 slots filled with useful operations). After vectorization, the number of cache stall cycles increases while execution time decreases. Therefore, those stalls cause the Efficiency to decrease.

To simulate the multi-threaded program, the simulator was configured with 9 Trimedia processors, which is the number of TriMedias targeted for the Wasabi chip. Figure 9 displays the execution time as a function of the number of threads.

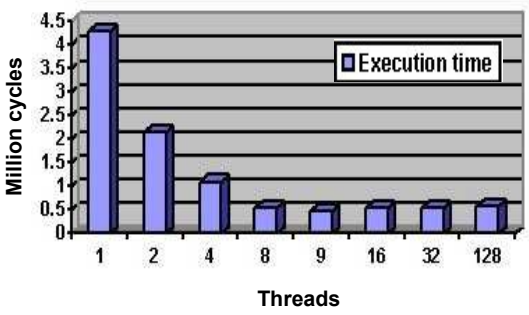

Figure 9: Execution time on 9 Trimedia processors as a function of the number of threads.

It can be seen from Figure 9 that the program scales well. When there are fewer threads than processors, the speedup is almost linear in the number of threads and, furthermore, when there are 9 threads, the speedup is almost a factor of 9x. Moreover, the amount of work per processor is very well balanced. Adding more threads could be beneficial if the work is unbalanced (because the scheduler assigns threads to processors that are idle) but here we see that the execution time increases slightly if more threads are added. This is because more threads imply more scheduling time.

Note however that Figure 9 is limited in

its semantics. The data to draw that figure was gathered in the same way as for the earlier figures, meaning that only the application load of the system is measured, excluding the overhead of other system tasks. In particular the system thread scheduler itself was not measured. However, Figure 9 clearly shows that (a) the application allows for plenty of parallelism for this multi-processor and (b) the system infrastructure and memory hierarchy are capable of providing sufficient to keep all processors effectively busy.

IV. MAPPING THE PARALLAX ALGORITHM ONTO WASABI

In this section, we describe a novel 3D-TV rendering algorithm [3], which generates a viewpoint-transferred image based on a primary image, and describe how that algorithm was mapped onto Wasabi.

Description of the algorithm

In Figure 10, the original and the viewpoint-transferred images are superimposed by making the original image half transparent. The image is blurred because the viewpoint-transferred image has a shift effect.

Figure 10: Superimposed effect.

Figure 11 shows a tree in front of a house with one scanline extracted.

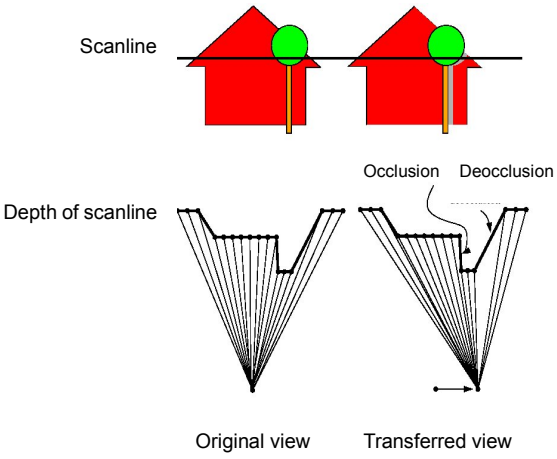

Figure 11: Rendering from image+depth: occlusions and deocclusions (example taken from [3, page 4]).

It is clear in Figure 11 that the depth map helps us to know the relative position of every pixel to the original viewpoint. Obviously, while rendering the image from the new viewpoint, the introduced occlusions should be dealt with. Moreover, the camera in the original viewpoint does not record some parts of the background house. (i.e., we lack information to reconstruct the image from the new viewpoint.) One option to solve that is to reconstruct extra information about the background by using a filter and to insert the reconstructed information at the position of deocclusion.

Implementation

The main steps of the algorithm are that (a) the occluded parts are simply discarded, (b) the extra information about background, which fills in the hole of deocclusion, is generated by a filter and (c) the occluding parts are reconstructed to the new Mapped_positions in the output image., which depends on how much the viewpoint is transferred. In the implementation, (b) and (c) are integrated into one routine. A pseudo-code description of this routine is

given in Figure 12.

```
Reconstructing( )
  Progressed _position=dx + t*subsegment;
  //t is recursive times of the routine subtrat 1
  //subsegment=(dn-dx)/(t+1).
  Mapped_position=
      integer around Progressed_position;
  Left=
(last_progressed_position + progressed_position)/2 -
Mapped_position;
  Right=
(next_progressed_position + progressed_position)/2 -
Mapped_position;
// Mapped_position is supposed to be the origin.
  do{
   W=filter_function(Right)-filter_function(Left);
   Out_R(mapped_position, y) += W * R(x,y);
   Out_G(mapped_position, y) += W * G(x,y);
   Out_B(mapped_position, y) += W * B(x,y);
   //Out_R, Out_G, Out_B are arrays storing
   //red green/ blue values for output image.
   Mapped_position++;
   Left - - ; Right - -;
  }while(W>0)
}
```

Figure 12: Pseudo-code of the Reconstructing routine.

Figure 13 illustrates the Reconstructing routine.

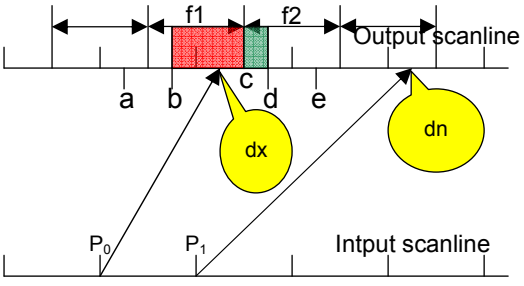

Figure 13: Illustration of the Reconstructing routine.
a:Last_progressed_position, b:Right,
c: Mapped_position d:Left,
e:Next-progressed-position.

In Figure 13, the pixel positions in both the input and output scanlines are consecutive integers. The occluding segment from pixel P_0 to P_1 is mapped to the segment from dx to dn in the output scanline. In the case illustrated by Figure 13, the do-while loop in Figure 12 is executed two times. The number of the loop times is guarded by the value of W. W is the red area ($W = \int_b^c f1$) in the first iteration and the dark pale area ($W = \int_b^c f2$) in the second. Consequently, the R, G, B values of P_0 are decomposed into Mapped_position and Mapped_position + 1. Furthermore, in the case shown in Figure 13, the whole Reconstructing routine is also computed two times. First, Progressed_position is dx (t=0) and, second, Progressed_position progresses to Next_progressed_position (t=1).

Optimizations

The following optimizations have been performed:

a) Simplification of the filter.
b) Algorithmic and arithmetic optimizations.
c) Vectorization.

The filters can be modeled as functions. Figure 14 shows three filters: the Box filter, the Tent filter and the Mitchell filter. In principle, the quality of the output image is positively correlated with the order of the filter function. It is obvious that high order filter functions are computationally more expensive. We therefore want to use a lower order filter that maintains the quality of the output image. We found that Using Box filter, the output image hardly depredates. Hereby, the Box filter is chosen and further optimizations are based on that.

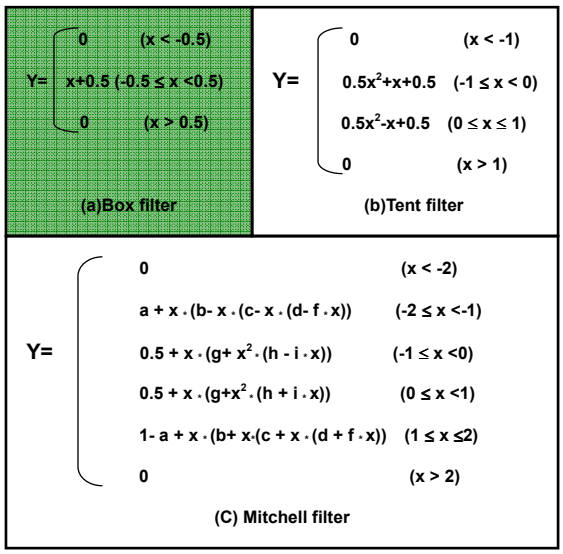

Figure 14: Filter functions. a,b,c,d,e,f,g,h **and** i **are all constants.**

The code profiling shows that the part of discarding occluded parts (step (a) of the algorithm) costs every little. Therefore, the main effort of algorithmic and arithmetic optimizations is only targeted at the Reconstructing routine.

From Figure 13, it can be seen that the Mapped_position always accumulates 1 or 0 but never negative values. Hereby, when the Mapped_position progresses, the pixel before it can be rendered out. As a result, in the actual implementation, register variables can be used to replace the arrays (Out_R, Out_G, Out_B in Figure 12). Consequently, the values in the registers are not written to memory until the Mapped_position proceeds. It implies that if the Mapped_position is the same, the values will stay in the registers, which saves writing the results to memory.

Moreover, because of the simplicity of the Box filter, the operation W = filter_function(Right)-filter_function(Left) can be implemented as W = Clip(-0.5, Right, 0.5) - Clip(-0.5, Left, 0.5). The call Clip(-0.5, x, 0.5) returns –0.5 if x is less than –0.5, 0.5 if x is greater than 0.5, and x otherwise.

In addition, the do-while loop in Figure 12 can be unrolled, because Right - Left will never be greater than 2 (i.e., the loop will be executed no more than 3 times.) Hence, we can unroll the loop for 3 times and, of course, add some extra corresponding modifications. Unrolling the loop eliminates many branches that the compiler generates. Consequently, that optimization reduces the branch delay penalty and speeds up the algorithm.

Figure 15 illustrates how the operations W*R(x,y), W*G(x,y), and W*B(x,y) in Figure 12 are vectorized.

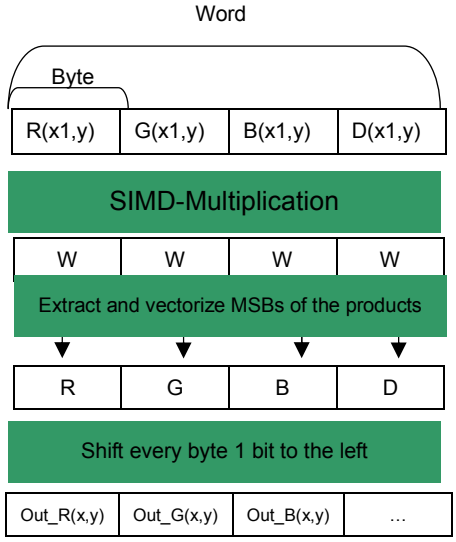

Figure 15: Vectorization.

In Figure 15, the word of Ws can come from a look-up table whose index is a concatenation of Right and Left. The primary value of W is in *[0,1]* but here every W is magnified by 128 times. The reasons to do that are (a) every W will not exceed 8 bits so that it can be vectorized, and (b) after the parallel multiplication, only the MSB of each product is extracted and vectorized. If W is not enlarged, accuracy

will be lost. Even though, every product has to shift 1 bit left at the end in order to keep the accuracy.

Multi-threading

In this algorithm, the arrays storing the normal image and its depth map cannot be distributed and divided flexibly because the computations are based on scanlines (fixed number of pixels). But we can process several scanlines in one thread. Therefore, the number of threads generated is equal to the number of scanlines divided by N. Obviously, when N=1 (i.e., each thread processes a single scanline), the multi-threaded implementation is much simpler. In addition, from Figure 9, it can be concluded that the overhead of the scheduler is rather small. Therefore, the performance results of the simple solution (N=1) can be representative of the results we need.

Experimental Results

All simulations for this algorithm were performed with 2 frames of 640x1080 pixels. Figure 16 depicts the execution time of the original code with different filters and that after the optimizations and vectorization (using the Box filter).

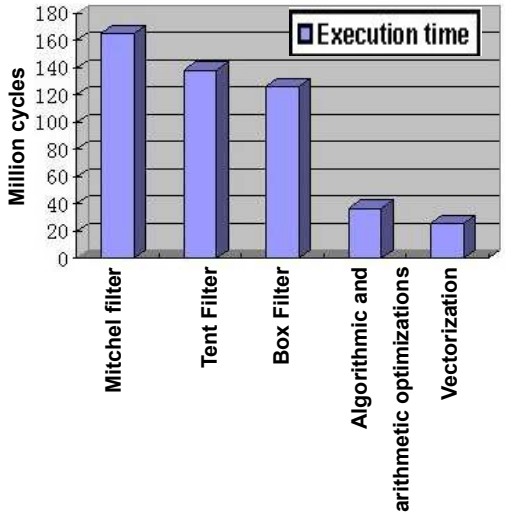

Figure 16: Execution time of the parallax algorithm on a single Trimedia processor.

It can be seen from Figure 16 that the performance improves by around 25% by simplifying the filter. And the algorithmic and arithmetic optimizations save additional 90 million cycles. Finally, vectorization speeds up the algorithm by another factor of 1.5x compared to the implementation after algorithmic and arithmetic optimizations. In total, the algorithm is accelerated by approximately a factor of 6.5x compared to the original code with the Mitchell filter. The resulting execution time is around 25 million cycles.

Figure 17 gives the number of cache stall cycles each program variant incurs. It shows that the number of cache stall cycles is almost unaffected by the filter that is employed. That is expected because changing the filter does not change the structure of the code. Algorithmic and arithmetic optimizations and again vectorization increases the number of cache stall cycles. The reason is partially that the cache stalls can no longer be hidden because the code is faster.

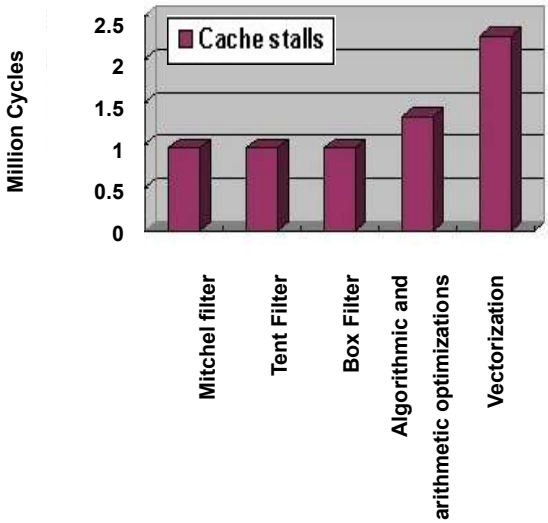

Figure 17: Number of cache stall cycles incurred by the different implementations of the parallax algorithm.

Figure 18 shows the Efficiency of the

different implementations of the parallax algorithm. Changing the filter does not affect the Efficiency either, because simplifying the filter does not only decrease the number of NOPs but also the number of effective operations. The increase of Efficiency after algorithmic and arithmetic optimizations is due to the reduction of control dependencies and converting floating point operations to integer ones. Consequently, the VLIWs contain more useful operations. After vectorization, the effect of cache stalls cannot be ignored any longer. Because the number of cache stall cycles increases relatively more while the number of execution cycles decreases, the Efficiency after vectorization is smaller than before vectorization.

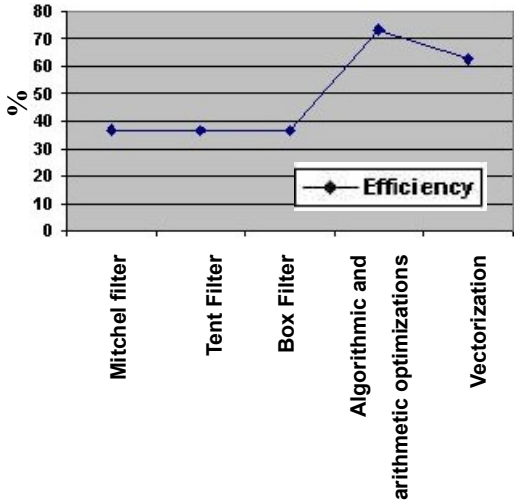

Figure 18: Efficiency of the different implementations of the parallax algorithm on a single TriMedia processor.

We now discuss the performance of the multi-threaded program. With 2 frames of 640x1080 input pixels, 1080 threads are generated. Again, the simulator was configured with 9 TriMedia processors. The resulting execution time is 3.8 million cycles. And the total workload of the 9 TriMedias is 27.9 million cycles. It can be seen that the total workload is roughly 9 times as much as the execution time. It implies that the workload on each processor is well balanced. Due to scheduling overhead, 9 times the execution time is slightly larger than the total workload. In addition, the total workload is also larger than the execution time obtained after vectorization on a single processor. That should also be attributed to scheduling overhead.

V. CONCLUSIONS AND FUTURE WORK

Given the cost of 3D-TV, Philips Research prefers to implement all 3D-TV applications on a single Wasabi chip. Consequently, the 3D rendering algorithm should occupy as few TriMedia processors as possible in order to leave processing time for some other 3D-TV applications.

The glassed-based algorithm perfectly matches this requirement. Even without exploiting thread-level parallelism, the algorithm runs in less than 5 million cycles on Wasabi. Assuming that the frequency of the TriMedia is 300MHz, one TriMedia processor is enough to render 60 pairs of output images per second, which matches the real-time requirement. Then, the other 8 TriMedias on Wasabi can be assigned to other applications. But people should wear a pair of special glasses to see the 3D effect and they get a common 2D effect by bare eyes.

Philips Research therefore advocates another algorithm - filtering occlusions and filling in deocclusions - whose 3D effect can be seen by our bare eyes. The output of this algorithm is a viewpoint-transferred image that is calculated from the original image and mixed by 3 basic colors (red, green and blue). Once we have that image, we can

generate 9 of such images, which are computed in different viewpoints and each of those images only contains one basic color. In the final rendering, 9 single color images with different viewpoints are integrated and interleaved into one frame and, then, that 9-view frame is projected on an LCD. Finally, through a special physical lens in front of the LCD, people will see the 3D effect.

Similar to the previous algorithm, the requirement is to render 60 of those 9-view frames per second, our experimental result -27.9 million cycles for one image with 3 basic colors- should, at least, be tripled. Therefore, 16 Trimedias or 2 Wasabi chip are needed to execute this algorithm in real-time. It seems to be expensive. So the future work is to develop another algorithm that is computationally less expensive or to optimize the parallax algorithm further, which is possible to design and develop some special hardware integrated in Wasabi to perform some parts of the algorithm.

REFERENCES

[1] P. A. Redert, Multi-viewpoint systems for 3D visual communication, Ph.D. Thesis, Delft University, 2000.

[2] P. A. Redert, System for Simultaneous 3D and Monoscopic Viewing, Philips research report, Natlab, TN 2002/312.

[3] R-P. Berretty and F. Ernst, Rendering Frames for 3D TV: Filtering Occlusions and Filling In Deocclusions, Philips research report, Natlab, TN 2003/109.

[4] P. Stravers and J. Hoogerbrugge, Homogeneous Multiprocessing and the Future of Silicon Design Paradigms, Proceedings of International Symposium on VLSI Technology, Systems, and Applications (VLSI-TSA), April 2001.