Soft Co-Processor Based Hardware Acceleration for Image Blending

Daniel Shapiro, Jonathan Parri, John-Marc Desmarais, Abdullah Kouri, Jean-Philippe Bergeron, and Miodrag Bolic

March 2011

Abstract The speed of software algorithms can be greatly improved by using a co-processor to offload computations from the main processor. Multiple co-processors can further increase the speed of a given algorithm. Based on this idea, three versions of an alpha blending algorithm were implemented on a NIOS II/f. The first implementation was entirely software based. This software based solution was then used as a baseline against which to test a single co-processor hardware solution and a multiple co-processor hardware solution. We showed that a single co-processor implementation achieved a speedup of 13.2 times, whereas the 2 co-processor solution achieved a speedup of 14.5 times with respect to this baseline. As further co-processors were added, the system became memory-bound as the algorithmic bottleneck moved from processing power to memory throughput.

1 Introduction

The Alpha Blending (AB) algorithm is a well known and widely used algorithm in image processing [1]. Due to its popularity, it is important to make this algorithm run more efficiently. The AB algorithm is used to transition between images in video editing. An example of one frame with a graded transition between two images is show in Fig. 1.

D. Shapiro, J. Parri, J.-M. Desmarais, A. Kouri, J.-P. Bergeron, and M. Bolic
School of Information Technology and Engineering, University of Ottawa,
Ottawa, Ontario
Tel.: +1-613-562-5800x2192
Fax: +1-613-691-1169
E-mail: {dshap092, jparr090, jdesm068, akour017, jberg081, mbolic} @site.uottawa.ca

In this paper, three optimizations are explored to obtain a speedup. First, since multiplication takes more time than bit shifting, the AB equation was modified to reduce two multiplications into one multiplication and one bit-shift operation. Second, the software algorithm was converted into hardware by writing an SOPC component in VHDL. The hardware component had the advantage of using hardware multipliers and hardware bound checking, whereas the selected processor did not use a hardware multiplier. Finally, multiple AB SOPC components were used in parallel to achieve an even higher speedup. At that point, a limitation of the AB system was discovered: as processing power is added, the application quickly becomes memory-bound. It was noted in [2] that accessing the main memory usually takes an order of magnitude longer than accessing local memory, and that this problem will become larger in the future. The application becomes memory-bound when four or more AB co-processors are used in parallel, thus no further improvements can be made by adding more co-processors. Other techniques such as using a wider memory bus or using a board with a faster memory are necessary to increase the speed beyond the presented solution.

2 Prior Art

As explained in [3], alpha blending is commonly performed in embedded systems such as DVD video post-processing. It is also used in 3D modeling as described in [4], 3D holographic imaging as in [5], VGA and LCD controllers as in [6], and in automotive video displays as in [7]. In terms of AB acceleration for the NIOS II/f processor, Xilinx and Altera have implemented proprietary AB accelerators using streaming protocols. [8,9].

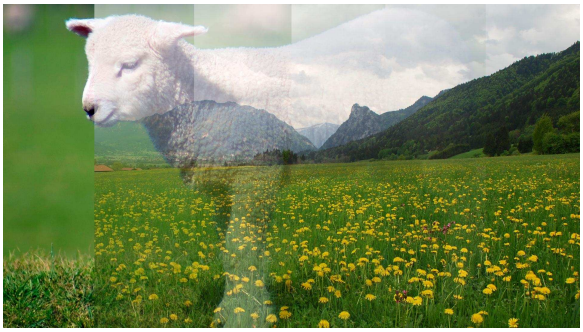

Fig. 1 Alpha blending example: An image of a sheep mixed with an image of a pasture.

Alpha Blending is a memory-intensive algorithm, that is, an algorithm which spends most of its processing time transferring data to and from the memory. As such, the efficiency of the AB algorithm is directly related to the efficiency of system memory[10]. Processing power is less essential to a memory-intensive system than memory throughput, thus after a small number of co-processors the system becomes memory-bound.

The efficiency of memory-bound applications can be improved by increasing the memory throughput of the system, by using other memory access techniques such as DMA, or by adding a cache. Another way to speed-up a memory-bound system is by using a larger memory word width. For example, nVidia's CUDA API implemented 64-bit and 128-bit load and store operations which can be executed in a single instruction[11]. Hence, more data can be processed for each store and load instruction. Since this solution requires more DMA transfers per instruction, there is an increased overhead in processing these instructions. In [12], the DMA overhead is discussed stating that there is a constant time required to queue a DMA transfer. This overhead may be overcome by sending more data per transfer. Therefore, a wider memory with wider data sets per DMA transfer can make the overall design more efficient. Interlacing accesses to memory banks can further reduce the latency of memory, where sequential addresses are stored in adjacent banks.

3 Experiment

Alpha blending is used to blend two images together into a single image, where the blend factor is modeled by the alpha variable. Usually, α is a floating point number between 0 and 1, but in this case, we used a blend factor or matte consisting of numbers between 0 and 255 because computers using an 8-bit colour palette tend to describe pixel values as being in this range and these integers are processed much faster by hardware than floating point numbers. As stated by [13]: "While the floating-point custom instructions are faster than software-implemented floating-point, they are slower than integer arithmetic."

Our algorithm starts by loading two images (160 x 120 pixels each) and a greyscale matte of the same size into local memory. The two input images are in 8-bit color, and hence they have 8-bit red, green, and blue color channels. Therefore, each pixel of the resulting image is the result of three AB computations; one blending operation on each colour channel. Each pixel of the matte determines the transparency for the three corresponding AB computations.

The AB algorithm is modeled by Eq. (1) [1].

$$V = \alpha_f V_1 + (1 - \alpha_f) V_2 \tag{1}$$

where $V_1(x, y)$,$V_2(x, y)$ are the pixels of the pictures to blend and $\alpha_f(x, y)$ is a value between zero and one representing the transparency of $V_1(x, y)$

From Eq. (1), we can derive a more efficient formula.

$$V = \alpha_f V_1 + V_2 - \alpha_f V_2 \tag{2}$$

$$V = \alpha_f (V_1 - V_2) + V_2 \tag{3}$$

Since we are loading matte values from an 8-bit grey scale image, we need to convert the grey scale data into a value between zero and one.

$$\alpha_f = \frac{\alpha}{256} \tag{4}$$

where $\alpha(x, y)$ is the 8-bit value loaded from the matte image.

Thus equation Eq. (3) becomes

$$V = \frac{\alpha(V_1 - V_2)}{256} + V_2 \tag{5}$$

Division by a power of two can be implemented using a bit-shift operation. We need to shift eight bits to the right to divide by 256, since $log_2(256) = 8$.

$$V = [\alpha(V_1 - V_2)] >> 8 + V_2 \tag{6}$$

As one may notice, Eq. (6) has one less multiplication and requires one shift operation. Since shifting can be accomplished by remapping wires in hardware with 0 clock cycle overhead, it is much more efficient than multiplication (multiple clock cycles), our hardware uses the algorithm of Eq. 6.

3.1 Hardware Setup

The overall architecture of the system can be seen in Fig. 2. The processor is the Nios II/f. The processor

schedules AB operations and operates the DMA controller, which transfers blocks of data from the SDRAM into the on-chip memory. The SDRAM contains all of the image data, i.e. foreground, background, matte, and result. Because the total on-chip memory of the Altera Cyclone II chip is 16kB [14], each AB component has an equal share of 4kB of local on-chip memory space. The DMA has access to all four on-chip memories. The AB components only interact with their own on-chip memory. Also, the processor has to initiate the computation by sending an address to an AB component. This component will then perform its computation starting at the supplied address. The DMA controller is also responsible for copying the data back to the main memory. Although not illustrated, a JTAG component is needed for transferring the images to the SDRAM and receiving the resulting alpha blended image to the Nios II IDE. A timer and a performance counter determine with high accuracy the time elapsed for the calculation. Finally, a PIO component was used for debugging purposes, in order to determine the state of each AB component.

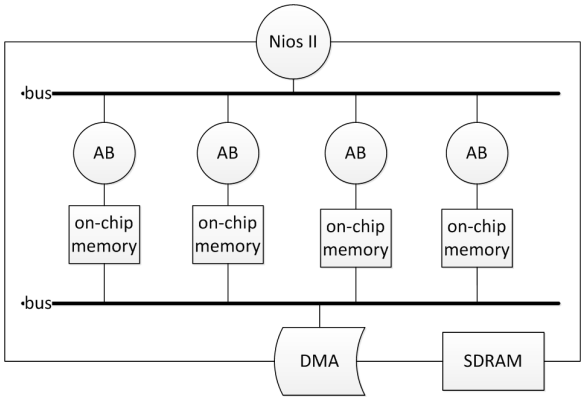

Fig. 2 Top level diagram of the system

The algorithm for AB as shown in Eq. (6) was implemented in hardware. A Finite State Machine (FSM) in Fig. 4 implements the AB algorithm. The main actions performed by each AB co-processor are to read an address from the memory, read two 8-bit integers starting at that address, Alpha-Blend three pixels packed in those two 8-bit integers, and store the result back into memory. States S1 through S15 are the states in our FSM. The transition to state S1 is initiated by the processor by sending a reset request. The system waits in S2 until the signal Write Byte (WB) becomes high, and this happens when an address is received. The address is stored in a register called "address", which represents the starting address where the AB will start. In S3, a request to the on-chip memory is made to the address from the register "address". In S5, when the signal Wait Request (WR) goes back to low, the value is stored in a register "short1" as seen in Fig. 3. The same steps are used for S6, S7 and S8 but to load another short from the memory to "short2" as seen in Fig. 3. It is necessary to load two 8-bit integers, because the memory is 16 bits wide and 24 bits of data are needed for an AB computation. An AB is done on two pixels from two images plus one pixel for the matte.

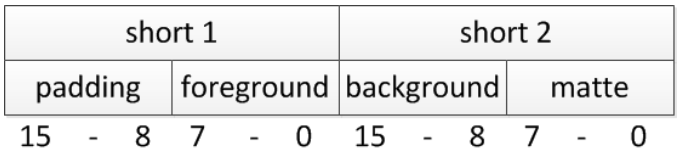

Fig. 3 Packing order of the pixels in an AB integer

Once the two 8-bit integers are loaded from the memory, the computations based on Eq. (6) can begin. The first step is at S9, which is to subtract the background from the foreground pixel. The second step in S10 is to multiply that resulting pixel by the matte pixel. It is important to note that the result of the subtraction is 9-bit and thus the result of the multiplication is 18-bit. The third step in S11 is to shift the resulting eight bits to the right to simulate a division by 256. Finally, in S12 the result is added to the background pixel and the operation is completed.

In the state S13, a request to the on-chip memory is made to the address from the register "address", to write the result from the AB computation. The system waits in the state S14, until the signal WR goes back to 0 to indicate that the write is complete. Finally in state S15, if the pixel counter is less than 1024 (our default block size), then the state returns to the state S3 where another computation can begin and the register "address" is increased by 4 to process the next pixel. The whole process is repeated until the pixel counter reaches 1024, where the AB component signals that it is finished.

3.2 Direct Memory Access

Since the AB algorithm must process data that is too large for the on-chip memory to store all at once, the images are stored on the SDRAM. We used Direct Memory Access (DMA) in order to reduce the overall transfer time from SDRAM to the on-chip memory and back. Our preliminary tests indicated that a simple memory transfer using pointers was much slower than a DMA transfer, because the DMA uses burst transfers. The

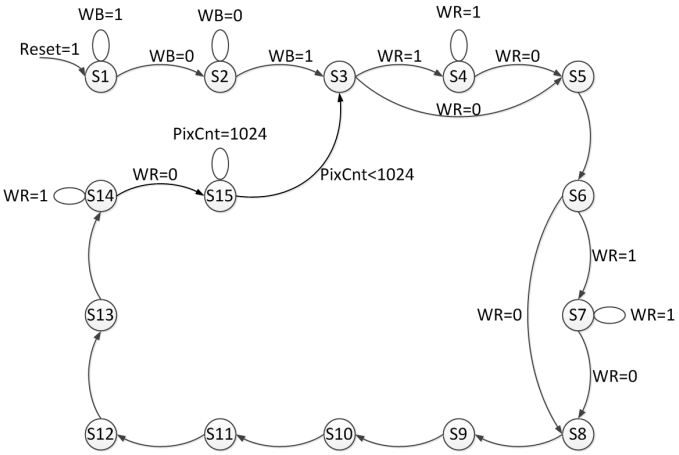

Fig. 4 Finite state machine diagram of the AB component

DMA transfers blocks of data as follows: From the C code, a command is issued to the DMA which requires the start and end addresses as well as the width of data to be transfered. This transfer will take data from the SDRAM and place it onto the on-chip memory of a given AB component. When the data is available on the on-chip memory, the AB component will begin the computation even during transfer. While it is transferring, the C code waits in a while loop until a flag indicating transfer complete is set. When this flag is set, we transfer the data back from the on-chip to the SDRAM using the start and end addresses of a portion of the new resulting image. After this transfer is done, another transfer begins with the next set of data.

If our on-chip memory and SDRAM had a larger bus width, for example 64-bit rather than the current 16-bit, we could send 4 times more data per transfer (64/16), and hence we would expect a speedup of 4 assuming enough AB components are used in the pipeline.

3.3 Image Conversion using C#

A C# program was used to prepare the images and help evaluate the results. This could have been done in the FPGA, but for simplicity, the whole task was accomplished in C# on the PC connected to the FPGA. This image preparation task between the PC and the FPGA was not counted in the speedup calculation since a real embedded system such as a hand-held video camera would not be fed data from a PC. Instead we could expect this data to be buffered directly into a memory.

3.3.1 Packing the images for alpha blending

Before loading the images from the SDRAM, each picture must be converted into a file for transferring. When the application is uploaded to the FPGA, we include an integer array in a C header file and it gets placed on the SDRAM. This header file is formatted using the bitmap pixels of the two images and the matte. Each pixel requires eight bits to cover all 256 possibilities. The size of an integer is 32-bits, and therefore we can store the foreground, background and matte pixels in the last 24 bits of an integer (see Fig. 3 for packing order). To create this header file, the C# code creates a file and reads in the image file, for each pixel we create an integer representing the three pixels saved in the packing order. Again, in a real video camera the optical sensor would buffer data directly into memory for processing by the AB components. We used performance counters to account for the print statements and initialization the would not happen on a real embedded camera platform.

3.3.2 Rebuilding the Resulting Image

After the AB is performed on the images, the resulting image is read from the SDRAM memory and printed out into a file on the connected PC. This file only contains bytes of data; therefore, we converted the bytes into an image using a C# program. The data received are red, green, and blue bytes that represent color channels for one pixel in the image. The C# code reads the image, and for every three bytes read a bitmap pixel is created which is attached to an image object. Finally, the image object is saved to the hard disk and we can visually observe the image.

3.4 Nios II/f Processor

The Nios II/f processor was used for synchronization and task delivery. The AB images were stored as an array of integers directly in the program's memory on the SDRAM. The AB program transfers a block of the image in the on-chip memory of one AB component and then starts the component. Another block of memory will be transferred to another AB component without waiting for the first component to complete. When the first AB component is done, the memory is transferred back to the SDRAM, and the program is free to facilitate computation on yet another block.

This pipelined algorithm can be seen in Fig. 5. The sizes of the blocks are taken from Table 1 and are proportional to the actual time of each action. In the first line, only one AB component is used and one can see that the DMA is not fully used, because it has to wait for the AB to be completed to transfer the data back. A speedup is obtained when two components are used, because a second DMA can begin while the first AB is

computing. The speedup is not optimal since the DMA is not fully used as seen in Fig. 5 next to the large black angled arrow. When 3 AB components are used, the speedup is increased but less than previously, because the DMA is always used and the algorithm becomes memory-bound. There is no advantage to using more than 4 AB components, because the DMA is the bottleneck. This result is not surprising since performance limitations tied to memory latency are common for many other algorithms [15,16].

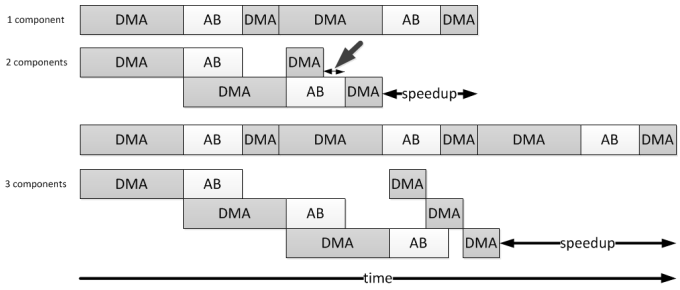

Fig. 5 Pipelining of DMA and AB components using (top to bottom) 1, 2, and 3 components.

Table 1 Time of the AB algorithm pipeline stage in clock cycle

Stage	clock cycle
DMA SDRAM to on-chip	23000
AB	12774
DMA on-chip to SDRAM	9513

3.5 Results and Discussion

The results of adding multiple AB components are shown in Tab. 2. One can see a speedup difference of 1.34 between 1 and 2 AB components. This is explained in Fig. 5 where most of the AB can be done in parallel with the DMA. From 2 to 3 AB components, an even smaller improvement in speedup is obtained, and this is explained in Fig. 5 where the DMA is always in use. After 4 AB components, no further speedup is achieved, because the DMA is always in use and the AB components are idle while waiting for data to arrive.

We converted the software implementation of (6) to a hardware implementation using an AB component. The baseline is the Nios II/f processor, executing a software implementation of AB as shown in Tab. 2. The hardware component performs over 10 times faster than the baseline. This speedup can is due to the fact that the multiplication uses embedded hardware multipliers and the data flow is in hardware.

Table 2 Comparison of approaches to performing alpha blending on 160 x 120 pixel color images.

	clock cycles	time (s)	speedup
Nios II/f software only approach	66297530	0.88397	-
Nios II/f with DMA and 1 co-processor	5025630	0.06701	13.19
Nios II/f with DMA and 2 co-processors	4563907	0.06085	14.53

4 Conclusion

This paper has found two optimizations for AB. First, the conversion from software to hardware has produced a speedup of 13.19. Second, parallelizing the processing was capped by the memory transfer speed of the DMA, thus a maximum speedup of 14.53 was achieved. Our design would scale perfectly assuming that enough AB components can fit on the board, thus the speedup of our design will match the maximum speed of the DMA.

References

1. Bo Shen, I.K. Sethi, and V. Bhaskaran. DCT domain alpha blending. In *Image Processing, 1998. ICIP 98. Proceedings. 1998 International Conference on*, volume 1, pages 857 –861 vol.1, oct 1998.
2. A. Cristal, O.J. Santana, F. Cazorla, M. Galluzzi, T. Ramirez, M. Pericas, and M. Valero. Kilo-instruction processors: overcoming the memory wall. *Micro, IEEE*, 25(3):48 – 57, june 2005.
3. Xiang-Yang Han and Di He. Co-deinterlacing between video decoder and video post-processor. In *VLSI Design and Video Technology, 2005. Proceedings of 2005 IEEE International Workshop on*, pages 353 – 356, may 2005.
4. E. Enderton, E. Sintorn, P. Shirley, and D. Luebke. Stochastic transparency. *Visualization and Computer Graphics, IEEE Transactions on*, PP(99):1, 2010.
5. Enrico Zschau, Robert Missbach, Alexander Schwerdtner, and Hagen Stolle. Generation, encoding, and presentation of content on holographic displays in real time. volume 7690, page 76900E. SPIE, 2010.
6. Richard Herveille. VGA/LCD Core v2.0 Specifications. rev. 1.2, Opencores.org, March 2003.
7. Mike Hutton and Roger May. Programmable Solutions for Automotive Systems. page 10, 2006. `http://sites.google.com/site/mhutton1/2006\_DATE\_MH\_auto.pdf` (accessed on Feb 10, 2011).
8. Altera Corporation. Alpha Blending Mixer MegaCore Function. page 6, 2006. `http://www.altera.com/literature/rn/rn\_alphamixer\_101.pdf` (accessed on Feb 9, 2011).
9. Reed P. Tidwell. Alpha Blending Two Data Streams Using a DSP48 DDR Technique. Technical Report XAPP706 (v1.0), Xilinx, March 2005.
10. Mustafa M Tikir, Laura Carrington, Erich Strohmaier, and Allan Snavely. A genetic algorithms approach to modeling the performance of memory-bound computations. In *Supercomputing, 2007. SC '07. Proceedings of*

the 2007 ACM/IEEE Conference on, pages 1 –12, nov. 2007.

11. J. Siegel, J. Ributzka, and Xiaoming Li. CUDA Memory Optimizations for Large Data-Structures in the Gravit Simulator. In *Parallel Processing Workshops, 2009. ICPPW '09. International Conference on*, pages 174 – 181, sept. 2009.
12. F. Khunjush and N.J. Dimopoulos. Extended characterization of DMA transfers on the Cell BE processor. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1 –8, april 2008.
13. Altera Corporation. Using Nios II Floating-Point Custom Instructions. page 7, 2008. `http://www.altera.com/literature/tt/tt\_floating\_point\_custom\_instructions.pdf` (accessed on November 27, 2010).
14. Altera Corporation. Using the SDRAM Memory on Alteras DE2 Board with Verilog Design. pages 3–9, 2008. ftp://ftp.altera.com/up/pub/Tutorials/DE2/Computer_Organization/tut_DE2_sdram_verilog.pdf (accessed on November 27, 2010).
15. S. Shida, Y. Shibata, K. Oguri, and D.A. Buell. Implementation of a barotropic operator for ocean model simulation using a reconfigurable machine. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 589 –592, aug. 2007.
16. S. Shida, Y. Shibata, K. Oguri, and D.A. Buell. An optimization method of DMA transfer for a general purpose reconfigurable machine. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 647 –650, sept. 2008.