

# Generalised Parallel Bilinear Interpolation Architecture for Vision Systems

Suhaib A. Fahmy  
 Centre for Telecommunications Value-chain Research  
 Trinity College Dublin  
 Ireland  
 suhaib.fahmy@tcd.ie

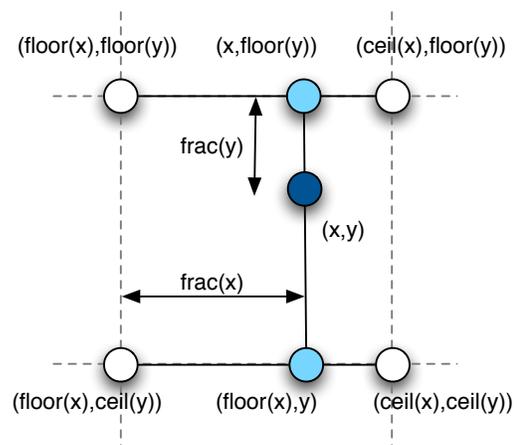
## Abstract

*Bilinear interpolation is widely used in computer vision for extracting pixel values for positions that lie off the pixel grid in an image. For each sub-pixel, the values of four neighbours are used to compute the interpolated value. This presents a challenge since four pixels must be read from the source image memory for each output pixel. This paper presents an architecture, for implementation within FPGA-based vision systems, that takes advantage of the heterogeneous resources available on modern devices to parallelise these memory accesses through efficient distribution of the source image in embedded memories. We show how intrinsic information in the sub-pixel addresses can be used to implement bilinear interpolation efficiently. We then suggest modifications to the architecture for larger image sizes which exceed the memory capabilities of modern FPGAs. The architecture is shown to achieve performance of 250Msamples per second in a modern device.*

## 1. Introduction

When applying computer vision techniques to an image, it is common to require some method by which to extract points that lie off the pixel grid. An algorithm used to transform an image, whether by rotation, shifting, zooming or the like, or access the image in an order or along contours that are not aligned with the source coordinates will produce sub-pixel sampling points for which values must be calculated by interpolating from the pixels in the source image.

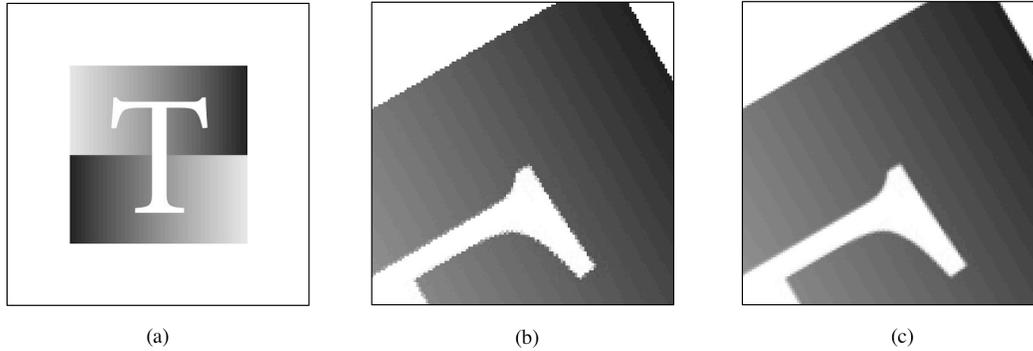
In the most basic case, the fractional part of any sub-pixel address is truncated or rounded, so each pixel simply takes the value of the nearest “real” pixel in the source image. This is called “nearest neighbour” approximation. This method is the simplest, and involves no calculation. It also only requires one pixel from the source image for each sub-pixel which is being calculated; hence it can operate at the full speed of the surrounding circuit. However, the resultant



**Figure 1. The bilinear interpolation grid.** Given a point  $(x,y)$ , we interpolate between  $\text{floor}(x)$  and  $\text{ceil}(x)$  on each of two rows  $\text{floor}(y)$  and  $\text{ceil}(y)$  using the fractional part of  $x$  to obtain two intermediate points shown. Then we use the fractional part of  $y$  to interpolate between these two points to obtain the final interpolated value.

approximation is not optimal with this technique. If used for image transformation, the deterioration in quality is clearly visible, while feature extraction can also be affected.

In bilinear interpolation, the value of a sub-pixel is interpolated from its four nearest neighbours linearly. The horizontal fractional component (of the sub-pixel coordinate) is used to compute two interpolated points that still lie on the horizontal grid, then the vertical fractional component is used to interpolate between these two points. Figure 1 shows how this is done for an image. In each case, the linear interpolation between two values  $a$  and  $b$  by fraction  $k$  is computed as  $(b - a) \times k + a$ . Bilinear interpolation offers significantly enhanced image quality over nearest-neighbour approximation [4]. Figure 2 shows a comparison between the two methods when rotating an image. Bilinear



**Figure 2. A close-up comparison of nearest neighbour approximation (b) and bilinear interpolation (c) when rotating a test image (a).**

interpolation suffers in speed terms because it requires access to four source pixels in order to compute a sub-pixel value. This typically means four memory accesses and thus a reduction in the speed of pixel extraction by a factor of four over nearest neighbour approximation, which results in slowing down the surrounding vision system [2].

Other interpolation methods exist, such as bicubic and sinc interpolation. While they offer even better quality in terms of the transformed image, the computational requirements are significantly increased. Hence, they are typically used for the final display of images, as opposed to the internals of a vision system, for which speed is of paramount importance. Bilinear interpolation is still widely used within vision systems. Note that while a system typically accepts input frames and produce output frames at standard rates, the processing rates within the system can be much higher. As an example, the Trace transform system in [3], processes 30 frames per second, but each of these frames accesses data in 180 transformed frames. Hence it is unfeasible to use external hardware, or even to access external memory within the processing pipeline.

FPGAs have found significant use in the field of image and video processing. A number of advanced techniques which otherwise map poorly to CPUs and GPUs can be accelerated by designing custom hardware architectures that exploit algorithmic parallelism. FPGAs also provide the ideal platform for testing and prototyping, given the ease with which a design can be modified. Many computer vision algorithms require pixel approximations, as described above, somewhere within the datapath. Indeed some of our previous work on the Trace transform [3] used nearest-neighbour approximation, and the design proposed in this paper could be integrated into that work.

Modern FPGAs provide a range of heterogeneous resources for use by the designer. Hard multipliers and DSP blocks are useful in a variety of signal processing applications, as are embedded memory blocks. These hard blocks offer guaranteed performance, while also freeing up logic

and routing resources for other parts of a design.

A typical application of bilinear interpolation within a vision system could become the bottleneck in speed terms due to the increased memory access requirements over nearest neighbour approximation. Even if a vision processing system is able to run at high speeds within the FPGA fabric through thorough pipelining, a standard sequential approach to bilinear approximation would reduce the speed to a quarter of its potential. The aim of this design is to compute one sub-pixel value in each clock cycle within a stream of images, and thus run as fast as the rest of the system..

In this paper we propose a generalised architecture for bilinear interpolation that can be integrated into any computer vision application implemented on FPGA. This architecture parallelises pixel accesses through distribution of the source image across embedded memories. It accepts a stream of sub-pixel addresses for which it computes the interpolated values in a single-cycle delay pipeline. This architecture can find use within the wide array of applications. We explore a method that can use embedded memories for images of small size, then propose how the architecture could be tweaked for processing images of large sizes, like Full HD (1920× 1080 pixels).

## 2. Previous Work

A standard technique for bilinear interpolation fetches each of the four required pixels for a sub-pixel address serially [2]. This means that four memory accesses are required per output pixel value. This can slow down processing significantly and preclude real-time performance. This is typical of solutions in the software world and considering the fast memory accesses and efficient caching schemes, is sufficient in such a context. In software, the interpolation calculation itself is also serial.

In cases where the interpolation factor is fixed, it is possible to apply a simple 2-dimensional filter to the image. This

might occur when an image is being expanded by a fixed factor, as in [9]. The author applies a  $4\times$  zoom ( $2\times$  in each dimension), by computing one interpolated pixel between each two adjacent pixels. Since the interpolated pixel will be half the sum of the two adjacent pixels, it is a relatively straightforward implementation.

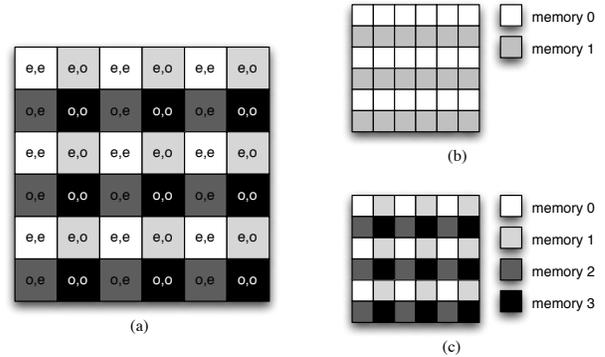
Hardware architectures for bilinear interpolation have typically not veered too far from the serial accesses mentioned, though pipelining the interpolation itself has been done before. In [6, 5], the authors develop an intelligent row caching scheme which maintains a cache of previously accessed pixels noting that they are reused in computing subsequent pixels. This allows a single access to memory to provide the required missing pixel. The proposed scheme is based on the assumption that the sub-pixel addresses are approximately spaced similarly to the source image pixel grid, or closer, and follow raster-scan order. Hence subsequent sub-pixels will have some neighbours in common, which can be cached. This holds true for most classes of transformation such as barrel distortion correction, to which they apply it. In some cases, where all four pixels cannot be obtained with a single memory access, a three-pixel interpolation is computed instead.

In order to allow for full parallel access to source pixels, a number of memory-based techniques can be employed. Consider the scenario where the source image is duplicated in four separate memories, each accessible in parallel. This would solve the problem of access but at the cost of increased memory requirements, due to the high level of redundancy. It is also possible to take advantage of multi port memories, but these are not widespread as distinct devices on development boards. Transferring data off-chip for processing can also result in a performance penalty.

It is worth noting that current state of the art for image *rendering* eschews bilinear, and even bicubic interpolation, in favour of filter based methods. H.264 and AVS video standards both make use of FIR filter based interpolation methods for rendering. These algorithms are being tackled by hardware designers [7, 11]. Bilinear interpolation is still used for sub-pixel extraction for computer vision tasks though [8], and hence an architecture that can be integrated with computer vision hardware, especially on streaming images, without adversely impacting performance is of great benefit.

### 3. Proposed Architecture

The proposed architecture is designed to address the following problem: given a piece of vision hardware that requires the computation of sub-pixel values for which it provides addresses, and a stream of input images, compute the sub-pixel values in real-time, in the order in which the addresses are provided, assuming that new images stream di-



**Figure 3. Mapping pixels in the source image to memories. The four combinations of pixel coordinates are shown in (a). A mapping for dual-port memories is shown in (b), while (c) shows a mapping for single-port memories, requiring 4 in total for full parallel access.**

rectly following each other, and the sub-pixel addresses provided for each image fall within the time it takes for the image to stream through. Hence for  $N \times N$  pixel images,  $N^2$  sub-pixel values can be computed at streaming real-time speed. Note that no assumptions are made about the order or pattern of sub-pixel addresses, which allows for complete freedom in defining pixel access schemes. For example, one could trace lines or contours in the image.

The proposed architecture works primarily by rearranging the streaming image pixels that enter the system into separate memories according to their coordinates, allowing for parallel access to the four required pixels for the interpolation. It is clear that the interpolation uses adjacent pixels in two subsequent rows of an image. Hence for parallel access, we require some way of accessing pixels in subsequent rows independently, as well as parallel access to adjacent pixels in each row. One method would be to store alternate rows in different memories; the even rows of pixels would be stored in one memory while the odd rows are stored in another. Then we would also require to store even and odd columns in different memories. Thus, we would have four memories which would contain pixels that fall into the following four groups of  $(x, y)$  coordinates: (even,even), (even,odd), (odd,even) and (odd,odd), shown in Figure 3. Now for any interpolation point, we could access the four memories in parallel to obtain the required pixels.

The embedded memories in modern FPGAs allow for a further optimisation; since these memories can be used in a dual-port configuration, it is possible to use only two memories. Storing alternate rows separately would still allow subsequent pixels from the same row to be accessed in parallel. This simplifies the architecture somewhat.

In order to allow the system to run uninterrupted, it is

necessary to implement a double buffer. We can assume that the input to the system is a continuous stream of pixels in raster scan format, for one image followed immediately by the next. In this case, it is necessary that we distribute the pixels for the first image amongst the memories on the first side of the double buffer, while the pixels from the second side of the double buffer are used by the interpolation engine. Once a whole image has been written to the first side, the roles are swapped, and the next image is written into the second side, while the first side is used for processing. Note that the sub-pixel addresses fed into the system will extract values for the previously stored frame.

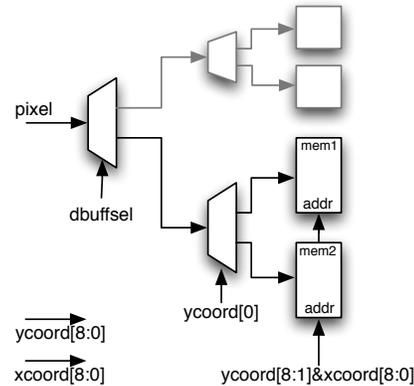
For the purposes of clarity, the remainder of this paper will deal with an implementation applied to a  $512 \times 512$  pixel 8 bits per pixel greyscale image. This assumption will be explored further in Section 3.2. We will also focus primarily on implementations using dual-port memories.

### 3.1. Addressing Scheme

In order to correctly select which buffer memory to write a pixel to, and which memories to read the source pixels from for interpolation, the pixel coordinates need to be manipulated. First, on the storage side, a row and column counter are kept up to date with each pixel that enters the system. The “new frame” pulse resets these so they maintain the correct values for each pixel. Since subsequent rows must be stored in different memories, we can use the least significant bit (LSB) of the row value to select between them. In the case of dual-port memories, this is sufficient. (If only single-port memories are available, then we must also select based on the LSB of the column value.) The rest of the row and column values (minus the LSB) gives the address at which to store the pixel in the relevant memory. This is shown in Figure 4

On the interpolation side, we must now consider how we interpret a sub-pixel address. Recall that we will always interpolate between subsequent rows and columns. However, it is possible that the upper row is odd or even, and similarly, the left column could be odd or even.

In the case of dual-port memories, adjacent pixels within the row can be accessed simply by passing two subsequent addresses to the two ports (while correcting the edge-cases to ensure the coordinates are valid). In the case of single-port memories we must also consider which memory is the leftmost. Firstly, we extract the integer parts of the sub-pixel address. The LSBs of these integer parts (x and y coordinates) determine whether the odd or even row or column are uppermost or leftmost. If the LSB is a zero, then we are interpolating between an even row and the following odd row. We pass the integer part of the address, minus the LSB to both (or, in the case of single-port memories, all four) memories, then maintain the order for addresses with zero LSB,



**Figure 4. One side of the double-buffer used for distributing incoming pixels between the two intermediate memories. The LSB of the y-coordinate selects which of the two memories a pixel is written to. (The two sides of the buffer are identical in the real circuit.)**

Address Bits	Interpretation
ycoord[15:8]	Row part of mem address
ycoord[7]	Which row is first in interpolation
ycoord[6:0]	Vertical interpolation factor
xcoord[15:7]	Column part of mem address
xcoord[6:0]	Horizontal interpolation factor

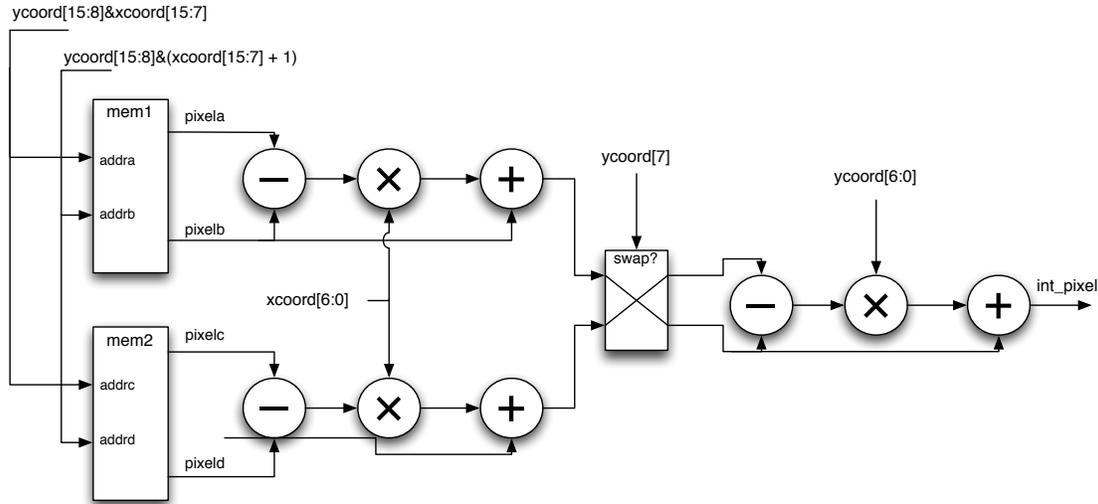
**Table 1. Interpreting the sub-pixel address for interpolation of  $512 \times 512$  pixel images.**

and swap them for addresses with a one LSB.

The fractional parts of each sub-pixel address are used for the linear interpolation further downstream. Since we now have access to all four pixels required in a single clock cycle, we can use a pipeline to interpolate, providing a full result in each clock cycle. The first part of the pipeline interpolates between the two adjacent pixels in each of the two rows (maintaining the ordering scheme mentioned above), by using the fractional part of the  $x$  coordinate, giving two row interpolation values. The next part of the pipeline interpolates between these two intermediate values, using the fractional part of the  $y$  coordinate to give the final interpolated value. Table 1 summarises the addressing scheme and Figure 5 shows the pipeline used for interpolation.

### 3.2. Alternative Considerations

While modern FPGAs contain significantly more memory on die than in the past, the amount available is still limited when considering the caching of images. Even when we consider a greyscale image of 8 bits per pixel (bpp), a



**Figure 5. The circuit for calculating the interpolated sub-pixel. Bit positions within the sub-pixel address are used as per Table 1. Two intermediate horizontally interpolated points are computed in parallel. The swap unit decides which is the uppermost pixel in the interpolation based on the LSB of the y-coordinate. The circuit is fully pipelined.**

512×512 pixel image would require 2.1Mbits of memory to store. A Full HD image at 1920×1080 pixels would require a full 16.6Mbits. Considering the need for double-buffering due to the real-time requirements of the system, these numbers must be doubled. Today’s most advanced, largest capacity devices from both Xilinx (Virtex 5) [10] and Altera (Stratix IV) offer at most 20Mbits of memory on chip. The memories available on devices are typically spread throughout the chip and thus suit integration within a larger vision system, but we must consider the limitations capacity places on our proposed architecture.

For situations where the on-die memory is insufficient, it is possible to adapt this architecture to use external memories. To simplify the implementation, ZBT SRAMs would be required, allowing pipelined single-cycle memory access. In such a system, the storage part of the architecture would store to external rather than internal memory. A double buffer would also be required. One difficulty is the number banks of memory required. Since external SRAM memory is usually single-ported, four memories would be required to allow for the parallel accesses, and further 4 for the double-buffer. One way this can be overcome, and bearing in mind the fact that SRAM memories are typically wide (>32 bits), is to concatenate neighbouring pixels in a single memory location. As an example, when storing the pixel for location  $(y_1, x_1)$ , we store both  $(y_1, x_1)$  and  $(y_1, x_1 + 1)$  concatenated. That way, when that location is read, the two neighbouring pixels are available immediately. This reduces the memory requirements to 4 external SRAM banks, which is more reasonable.

Concatenating unrelated data which is accessed using an identical pattern is an efficient way of parallelising. In [3] we showed how concatenating four orthogonal rotations within one memory location could provide four rotations of an image in the time normally taken for a single rotation.

The performance of such a system, based on external memory, is clearly dependent on the speed of the memory banks. The Celoxica RC300 board has 6 SRAM banks, accessible at speeds of around 80MHz. This would translate to 80 Mpixels/second. Faster boards exist today, but not having access to such hardware, we cannot confirm the precise performance on such boards. The I/O connectivity requirements are in line with many development boards in production.

Some other solutions, when dealing with specialised problems can also be investigated. If the sub-pixel addresses are requested in some predictable order, and are sequentially local, then it would be possible to process parts of the image in separate runs. For example, if applying image zooming, subsequent pixels will still be in the same rows, so we could process blocks of rows together. The architecture we present here makes no assumptions about the sub-pixel addresses being requested and thus maintains full flexibility for the vision system making use of it.

## 4. Results

The architecture was implemented in VHDL, making use of Xilinx CoreGen memory cores in order to provide maximum performance. The circuit was synthesised, mapped,

Resource Type	Amount (%)
Slices	362 (1%)
18Kbit Block Memories	256 (80%)
DSP48 Blocks	3 (1%)
Minimum clock period	3.994ns

**Table 2. Synthesis results for a Xilinx Virtex 4 XC4VSX55.**

placed and routed using Xilinx ISE 10.1. The target device was a Xilinx Virtex 4 XC4VSX55 FPGA. The SX family includes a higher proportion of embedded memories as required for this application. As has been mentioned above, the logic requirements for the architecture are extremely minimal, while the memory requirements are substantial for an FPGA application. Considering the proposed context for this architecture, this platform is ideally suited, since a significant portion of the logic fabric, a large number of embedded DSP blocks, and a reasonable number of memory blocks are available for the implementation of the vision portion of a complete system.

In order to extract maximum performance from the design, it was highly pipelined, including at the inputs and outputs to the embedded memories. Since the embedded Block Memories are small (holding 18Kbits each), a single system-level memory is in fact distributed throughout the device fabric. Hence routing can become a problem if the memory outputs are not registered. We found that the unregistered version of the design had a maximum clock speed of 133MHz. After registering, the clock speed increased to 250MHz, or 250 Msamples/sec.

The resource requirements are summarised in Table 2. At 250MHz, this architecture can process  $512 \times 512$  pixel images at speeds equivalent to 950 frames per second. The memory requirements of this architecture vary linearly with image size. The largest Virtex 5 device, the XC5VSX240T, contains sufficient memory for processing  $1024 \times 1024$  pixel images. Clearly, for larger images that would require external memory, the resultant speed would depend on the maximum access speeds for the external memory. For comparison against software, we consider a tool called O-Matrix [1], that is used in industrial applications. It exhibits a huge performance advantage over MATLAB for image processing functions. For image rotation<sup>1</sup> with bilinear interpolation, they claim performance of 15ms (as opposed to MATLAB's 2000ms) for a  $1280 \times 960$  image (equivalent to 8.2Mpixels/second). Our system is over 30 times faster. We must also factor in that for more complicated systems the speedup over software will be more significant, since

<sup>1</sup>Image rotation is chosen as it is a very simple operation which uses interpolation.

the computer vision part will be significantly faster.

The system was tested by preparing serialised input files of image data and sub-pixel coordinates in MATLAB and using these as the input to a VHDL testbench.

## 5. Conclusion

We have presented a generalised architecture for bilinear interpolation for use within vision applications implemented on FPGAs. The architecture distributes source image pixels across multiple memories which are then accessed in parallel. By using intrinsic data in the sub-pixel addresses, interpolations can be computed using minimal hardware resources, freeing up most of the device's logic fabric and embedded DSP blocks for implementing the remainder of the vision system. The architecture achieved a clock speed of 250MHz on a Xilinx Virtex 4. We also suggested how the architecture could be modified for use with larger images that exceed the available memory on the FPGA.

## References

- [1] O-matrix webpage (<http://www.omatrix.com/benchipt.pdf>).
- [2] J. Baltes and J. Anderson. Interpolation methods for global vision systems. In *RoboCup 2004: Robot Soccer World Cup VIII*, 2004.
- [3] S. Fahmy, C.-S. Bouganis, P. Cheung, and W. Luk. Real-time hardware acceleration of the trace transform. In *Journal of Real-Time Image Processing*, volume 2, pages 235–248, 2007.
- [4] R. Gonzalez and R. Woods. *Digital Image Processing*. Pearson Prentice Hall, 2007.
- [5] K. Gribbon and D. Bailey. A novel approach to real-time bilinear interpolation. In *IEEE International Workshop on Electronic Design, Test and Applications*, 2004.
- [6] K. Gribbon, C. Johnston, and D. Bailey. A real-time FPGA implementation of a barrel distortion correction algorithm with bilinear interpolation. In *Image and Vision Computing New Zealand*, pages 408–413, 2003.
- [7] Z. Haihua, X. Zhiqi, and C. Guanghua. VLSI implementation of sub-pixel interpolator for h.264/avc encoder. In *International Symposium on High Density Packaging and Microsystem Integration*, 2007.
- [8] F. Kelly and A. Kokaram. Fast image interpolation for motion estimation using graphics hardware. In *Proceedings of the SPIE*, pages 184–194, 2004.
- [9] P. Uthaichana and E. Leelarasamee. A pipelined bilinear interpolation for real time video image expansion. In *IEEE Region 10 Conference*, 2004.
- [10] Xilinx Inc. *Virtex-5 User Guide*, 2007.
- [11] D. Zhou and P. Liu. A hardware-efficient dual-standard vlsi architecture for mc interpolation in avs and h.264. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2007.