



# Image Processing The Easy Way

Image Zoom with and without the AMD Performance Library

Brent Hollingsworth  
Advanced Micro Devices  
November 2006

**2006 Advanced Micro Devices, Inc.** All rights reserved.

The contents of this document are provided in connection with Advanced Micro Devices, Inc. ("AMD") products. AMD makes no representations or warranties with respect to the accuracy or completeness of the contents of this publication and reserves the right to make changes to specifications and product descriptions at any time as the information contained herein may be of a preliminary or advance nature and is subject to change without notice. No license, whether express, implied, arising by estoppel or otherwise, to any intellectual property rights is granted by this publication. Except as set forth in AMD's Standard Terms and Conditions of Sale, AMD assumes no liability whatsoever, and disclaims any express or implied warranty, relating to its products including, but not limited to, the implied warranty of merchantability, fitness for a particular purpose, or infringement of any intellectual property right.

AMD's products are not designed, intended, authorized or warranted for use as components in systems intended for surgical implant into the body, or in other applications intended to support or sustain life, or in any other application in which the failure of AMD's product could create a situation where personal injury, death, or severe property or environmental damage may occur. AMD reserves the right to discontinue or make changes to its products at any time without notice.

## Table of Contents

Introduction.....	4
Image Zoom.....	4
Figure 1. Zoom image example .....	4
Image Zoom Algorithm .....	4
Figure 2. Zoom algorithm example .....	4
Figure 3. Resize calculations for dst(0,0) through dst(3,3).....	5
Image Zoom Implementation.....	6
Figure 4. Zoom implementation .....	6
Image Zoom Implementation – Multithreading.....	6
Figure 5. Threading data and entry point.....	6
Figure 6. Thread data partitioning .....	7
Image Zoom – A Better Way.....	8
Figure 7. APL implementation of image zoom .....	8
Figure 8. Function time by implementation.....	8
Conclusions - The Money Value of Time.....	9

## Introduction

In the recent article “Man on a Mission” (<http://www.devx.com/amd/Article/32312>), I introduced the AMD performance library as an answer to a parallel programming challenge. With this article, however, I’d like to take a very different approach. I will guide the reader through the manual implementation and multithreading of a common image processing routine: image zoom. I’ll describe the algorithm, provide the code, and then I’ll demonstrate a much better way to do it.

## Image Zoom

Conceptually image zoom is very simple. The left portion of Figure 1 contains a small selected area. We refer to this selection as the *region of interest* or ROI. In the zoom operation, the region of interest is stretched to fill the entire buffer. The image to the right shows the result of zooming into a 500x500 pixel area of a 5 megapixel image.

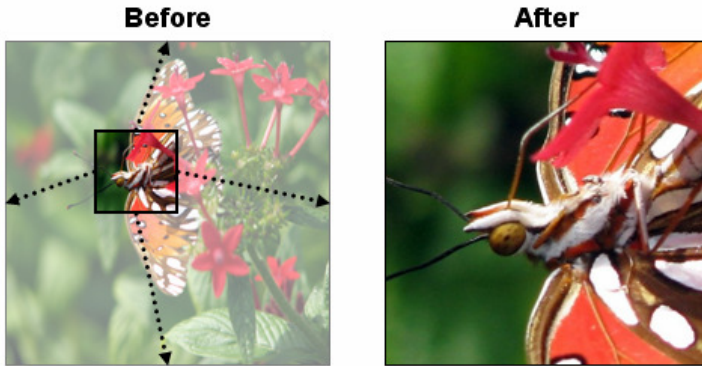


Figure 1. Zoom image example

## Image Zoom Algorithm

The implementation of image zoom is also quite straightforward. Consider the 6x6 pixel source image of Figure 2. We will zoom in on the region between (3, 2) and (5, 4) to produce the 6x6 pixel destination image on the right.

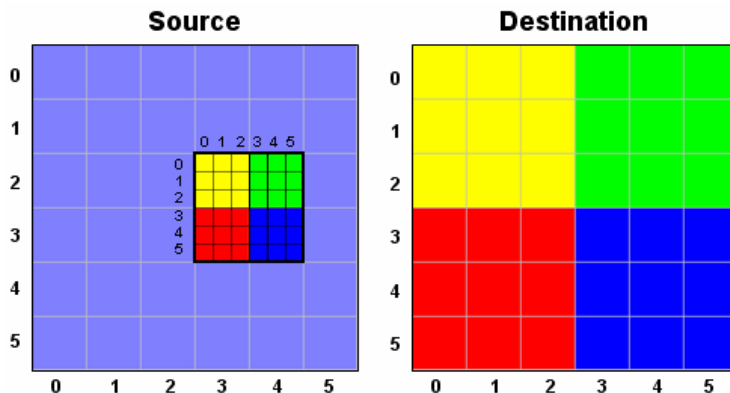


Figure 2. Zoom algorithm example

There are four basic steps to the algorithm.

**1. Calculate x\_ratio and y\_ratio**

Two columns in the source image will be stretched to fit six columns in the destination. This gives us a sampling ratio of 1/3 in X. In this example, the calculation is the same for X and Y.

$$x\_ratio = y\_ratio = \frac{2 \text{ pixels of source}}{6 \text{ pixels of destination}} = 1/3$$

**2. Calculate the coordinates of the source pixel corresponding to destination(0,0)**

With truncation sampling, the value for a destination pixel (x, y) is always obtained by copying directly from a source pixel (x', y'). x' is determined by multiplying x by the x\_ratio and adding the x offset of the region of interest.

Thus for destination( 0, 0 ):  $x' = 0 * .333 + 3 = 3$   
 $y' = 0 * .333 + 2 = 2$

**3. Fill destination( 0, 0 ) with source( x', y' )**

Step 2 produced x'=3 and y'=2. Therefore destination( 0, 0 ) = source( 3, 2 ) = Yellow.

**4. Repeat steps 2 and 3 for all other (x, y) in the destination**

We now step through the remaining pixels in the destination image, calculating the source coordinates, and copying from the source to the destination. In general, we will use the formula:

$$\begin{aligned} \text{destination}( x, y ) &= \text{source}( x', y' ) \text{ where} \\ x' &= \text{floor}( x * x\_ratio + x\_offset ) \\ y' &= \text{floor}( y * y\_ratio + y\_offset ) \end{aligned}$$

Figure 3 shows the calculated values for the destination range between (0,0) and (3,3) before the floor() is applied. Note that the call to floor() reduces every calculation in Figure 3 to (3,2). Yellow will be copied to each destination.

$x' = 0 * .333 + 3 = 3$	$x' = 1 * .333 + 3 = 3.333$	$x' = 2 * .333 + 3 = 3.333$
$y' = 0 * .333 + 2 = 2$	$y' = 0 * .333 + 2 = 2$	$y' = 0 * .333 + 2 = 2$
$x' = 0 * .333 + 3 = 3$	$x' = 1 * .333 + 3 = 3.333$	$x' = 2 * .333 + 3 = 3.666$
$y' = 1 * .333 + 2 = 2.333$	$y' = 1 * .333 + 2 = 2.333$	$y' = 1 * .333 + 2 = 2.333$
$x' = 0 * .333 + 3 = 3$	$x' = 1 * .333 + 3 = 3.333$	$x' = 3 * .333 + 3 = 3.999$
$y' = 2 * .333 + 2 = 2.666$	$y' = 2 * .333 + 2 = 2.666$	$y' = 2 * .333 + 2 = 2.666$
$x' = 0 * .333 + 3 = 3$	$x' = 1 * .333 + 3 = 3.333$	$x' = 3 * .333 + 3 = 3.999$
$y' = 3 * .333 + 2 = 2.999$	$y' = 3 * .333 + 2 = 2.999$	$y' = 3 * .333 + 2 = 2.999$

Figure 3. Resize calculations for dst(0,0) through dst(3,3)

## Image Zoom Implementation

Figure 4 gives a C++ implementation for the algorithm described above. Please note two changes:

1. The source pointer now points directly to the region of interest, eliminating the need to add a source offset.
2. The function takes two arguments, *dY\_start* and *dy\_end* which describe a vertical range on which to operate. This will be useful later for multithreading.

```
void Zoom( char *src, char *dst, int img_width,
           int dY_start, int dY_end,
           double x_ratio, double y_ratio )
{
    for( int dY=dY_start; dY<dY_end; ++dY ) // For each destination row.
    {
        int sY = (int)(dY * y_ratio); // Find the source row.
        char *sRow = src + sY * img_width * 3; // Point to the start of the row.
                                                // 3 is the pixel size in bytes.
                                                // For each destination column.
        for( int dx=0; dx<img_width; ++dx )
        {
            int offsetX = (int)(dx*x_ratio)*3; // Find the source row offset.
                                                // 3 is the pixel size in bytes.
                                                // Copy color from src to dst.
            *dst++ = sRow[ offsetX + 0 ];
            *dst++ = sRow[ offsetX + 1 ];
            *dst++ = sRow[ offsetX + 2 ];
        }
    }
}
```

Figure 4. Zoom implementation

I ran this function on the five megapixel image displayed in Figure 1. It ran in 226.67 million processor cycles, for an average of 44.99 cycles per pixel.

## Image Zoom Implementation – Multithreading

As a first step in threading, I create a structure *ZoomArgs*. This will hold the arguments for my zoom function. I also create a thread entry point *Zoom\_Thread*. See Figure 5.

The *Zoom\_Thread()* function casts the passed void pointer into a *ZoomArgs* pointer, then calls the original *Zoom()* function. Using this method greatly simplifies thread creation. By passing arguments in this manner, I can use my original *Zoom()* function (Figure 4) in my threaded implementation.

```
struct ZoomArgs
{
    char * src, * dst; // Pointers to buffers.
    int dY_start, dY_end; // Rows to process in thread.
    int img_width; // Image width.
    double x_ratio, y_ratio; // Ratios of src to dst.
};

DWORD WINAPI
Zoom_Thread( void * param )
{
    ZoomArgs * p = (ZoomArgs*) param; // Retranslate the void *.
    Zoom( p->src, p->dst, p->img_width, // Call the Zoom function.
          p->dY_start, p->dY_end,
          p->x_ratio, p->y_ratio );
    return 0;
}
```

Figure 5. Threading data and entry point

With the argument structure and entry point in place, I have only to write my threading function. This will create threads, divide and assign the work, and wait for threads to complete their job. The implementation is given Figure 6.

```

void Zoom_T( char * src,      char * dst,
             int  img_height, int  img_width,
             int  x_start,   int  y_start,
             int  x_end,     int  y_end )
{
    const int threads = 2; // Number of threads to use.
    ZoomArgs args [ threads ]; // Argument storage for threads.
    HANDLE handles[ threads ]; // Handle to each thread.

    double x_range = x_end - x_start;
    double y_range = y_end - y_start;
    double x_ratio = x_range / img_width; // Ratio by which to reduce X.
    double y_ratio = y_range / img_height; // Ratio by which to reduce Y.
    int step = img_width * 3; // Image width in bytes.
                                // 3 bytes per pixel.

    src += (y_start*step) + (x_start*3); // Move to the region of interest.
    div_t dY_block = div( img_height, threads ); // Divide the rows by the threads.

    int dY_start = 0; // Calculate the first block size.
    int dY_end = dY_block.quot + dY_block.rem;

    for( int t=0; t<threads; ++t ) // Assign data to each thread.
    {
        args[t].src = src;
        args[t].dst = dst + dY_start * step;
        args[t].img_width= img_width;
        args[t].dY_start = dY_start;
        args[t].dY_end = dY_end;
        args[t].x_ratio = x_ratio;
        args[t].y_ratio = y_ratio;

        handles[t]= CreateThread(0,0,Zoom_Thread, // Create a worker thread.
                                args+t,0,0);
        dY_start = dY_end; // Prepare the next offsets.
        dY_end += dY_block.quot;
    }
    ::WaitForMultipleObjects( threads, handles, // wait for threads to complete.
                             TRUE, INFINITE );
}

```

**Figure 6. Thread data partitioning**

I ran this code on my two processor system and measured a time of 113.86 million cycles for an average of 22.60 cycles per pixel. This is twice as fast as the non-threaded code!

## Image Zoom – A Better Way

Finally, after much effort we come to the better way to implement image zoom - get someone else to do it. The AMD Performance Library (APL) team has genius engineers like Wei-Lien Hsu “Mr. Multimedia” (<http://www.devx.com/amd/Article/32369>). Let them do all the hard work!

To perform image zoom using APL, I call `apliResize_8u_C3R()` after performing a minimal amount of calculation. See Figure 7 for the APL based implementation of image zoom. Note that the APL function uses Nearest Neighbor sampling which is superior to my implementation.

```

void Zoom_APL( char * src,      char * dst,
               int  img_height, int  img_width,
               int  x_start,   int  y_start,
               int  x_end,     int  y_end )
{
    int step      = img_width * 3;           \\ calculate the step size.
    int x_range   = x_end - x_start;
    int y_range   = y_end - y_start;

    ApliSize srcSize = {img_width, img_height}; \\ Source image size.
    ApliSize dstSize = {img_width, img_height}; \\ Destination image size.
    ApliRect srcRoi  = {x_start, y_start,      \\ Specify the region of interest.
                       x_range, y_range };

    double x_factor = (double)img_width / x_range; \\ Resize factors in X and Y.
    double y_factor = (double)img_height / y_range;

    apliResize_8u_C3R( \\ Call the APL function.
        (Apl8u*)src, srcSize, step, srcRoi,
        (Apl8u*)dst, dstSize, step, dstSize,
        x_factor, y_factor, APLI_INTER_NN ); \\ Use Nearest Neighbor sampling.
}

```

Figure 7. APL implementation of image zoom

I ran this implementation and obtained a time of 70.73 million processor cycles for an average of 14.04 cycles per pixel.

Figure 8 shows the average performance for each implementation.

	Average Cycles	Cycles/Pixel
<b>Zoom</b>	226.67 Million	44.99
<b>Zoom_T</b>	113.86 Million	22.60
<b>Zoom_APL</b>	70.73 Million	14.04

Figure 8. Function time by implementation

## Conclusions - The Money Value of Time

I invested nearly eight hours developing my Zoom application, including design, debugging, and optimization. The APL portion of the code took less than one hour to write, runs in 38% less time, and produces a higher quality result.

So why did the APL code beat mine so soundly?

- APL uses SIMD instructions such as SSE2 and SSE3 which perform operations like add and multiply in parallel. Adding a SIMD implementation to my Zoom function would require more time than I could afford to spend. It also requires advanced programming techniques which go far beyond standard C++ development.
- Managing threads can be very expensive. APL uses advanced thread pooling which avoids the overhead of calls like `CreateThread()`.
- APL developers meticulously tune their code for optimum performance.

The next time you are faced with a multimedia programming task, check out the AMD Performance Library. It can significantly reduce the burdens of debugging, multithreading, and optimization.

For more information, check out <http://developer.amd.com/apl.jsp>.