Accelerated 2D Image Processing on GPUs

Bryson R. Payne[1], Saeid O. Belkasim[2], G. Scott Owen[2],
Michael C. Weeks[2], and Ying Zhu[2]

[1] Georgia College & State University, Department of ISCM, Milledgeville, GA 31061
bryson.payne@gcsu.edu
[2] Georgia State University, Department of Computer Science, Atlanta, GA 30303
{sbelkasim, sowen, mweeks, yzhu}@cs.gsu.edu

Abstract. Graphics processing units (GPUs) in recent years have evolved to become powerful, programmable vector processing units. Furthermore, the maximum processing power of current generation GPUs is roughly four times that of current generation CPUs (central processing units), and that power is doubling approximately every nine months, about twice the rate of Moore's law. This research examines the GPU's advantage at performing convolution-based image processing tasks compared to the CPU. Straight-forward 2D convolutions show up to a 130:1 speedup on the GPU over the CPU, with an average speedup in our tests of 59:1. Over convolutions performed with the highly optimized FFTW routines on the CPU, the GPU showed an average speedup of 18:1 for filter kernel sizes from 3x3 to 29x29.

1 Introduction

Programmable graphics processing units (GPUs) are commonly included as hardware components in new computer workstations. Furthermore, the current generation of GPU is technically superior in terms of GFLOPs of processing power [7] to that of current CPUs [2], but that power is rarely used to its full capabilities. Numerous advances have been made recently in applying the GPU to parallel matrix processing tasks, such as the Fast Fourier Transform [6]. This research applies the vector processing power of the GPU to 2D image processing convolution filters.

Section 2 briefly covers recent GPU advances, and discusses why the GPU is well suited for digital image processing applications. Section 3 explains how convolution on images can be achieved with a GPU, while section 4 compares the frames-per-second processing rates of an NVIDIA GeForce FX6800 to a dual 2.8 GHz Intel Pentium 4. Section 5 notes our conclusion of an average 18:1 speedup of the GPU over a CPU running optimized code on convolution filter mask sizes 29x29 and smaller, and a speedup of 59:1 over a CPU without optimized code.

2 Background

The first programmable consumer GPUs became available less than four years ago. The first generation of programmable GPUs was not well-suited to general-purpose

V.S. Sunderam et al. (Eds.): ICCS 2005, LNCS 3515, pp. 256 – 264, 2005.

computation for several reasons. First, they allowed access to only selected portions of the graphics pipeline and had no easily accessible off-screen rendering capabilities. Second, they had to be programmed in GPU-specific assembly code, with no standardization across manufacturers. Third, their limited accuracy of 8 bits-per-pixel combined with their slower clock speeds and memory accesses, as well as smaller memory sizes, compared to CPUs made them unattractive to general-purpose computing researchers who needed fast 32-bit floating point operations as a minimum point of entry.

2.1 Computation on GPUs

By late 2002, a C-like programming language for the GPU, named Cg [5], had been developed for cross-platform GPU programming. Cg contained constructs for looping and conditional branching, required for most general-purpose computing, but GPU hardware took another two years to catch up to the capabilities provided for in the Cg language. Only in the NVIDIA GeForce FX6800 series GPU, released in late 2004, was it first possible to take advantage of true conditional branching on the GPU, as well as handle loops or programs that consisted of more than 1024 total instructions per pixel [7]. These advances enabled the research in this paper, and the previous work [8] upon which it is based.

Researchers at Stanford developed Brook for GPUs [1], a stream processing language abstraction for GPU programming. Unfortunately, stream processing is not especially well-suited to the limitations of current GPU hardware. Even the simple merge sort, a typical example of stream processing computation, is not possible in a single pass on current GPU hardware due to the inability to both read and write the same texture at the same time. This factor, along with others, caused Brook to show only a limited speedup on a number of test applications across platforms and manufacturers.

2.2 Digital Image Processing

Digital image processing (DIP) appears to be especially well-suited to current GPU hardware and APIs (application programming interfaces), due to the graphical nature of the GPU's processing power. Digital image processing is a field of computing that involves manipulating images that have been converted into digital form. Typical examples of DIP operations include smoothing or blurring an image, sharpening an image, edge detection, noise removal, and various effects such as embossing or beveling.

Figure 1 shows an original image with three types of filters applied. In addition to single 2D image filtering applications like those shown in Figure 1, DIP can be applied to video on a frame-by-frame basis, such as for motion detection. They can also be applied to 3D scenes generated on the GPU, such as when a flight simulator uses blurring to make a scene more realistic.

Many image processing operations are convolution-based, which means that every pixel of an image is processed by multiplying the elements of a smaller matrix, called a mask, with the values of the surrounding pixels, known as a neighborhood. For example, in a 3x3 averaging filter, the values of the nine pixels that comprise a pixel and its eight surrounding neighbors within a one-pixel boundary are summed together and the result is divided by nine for the output pixel's value. This type of filter smoothes or blurs rough edges in an image, like the first example given in Figure 1.

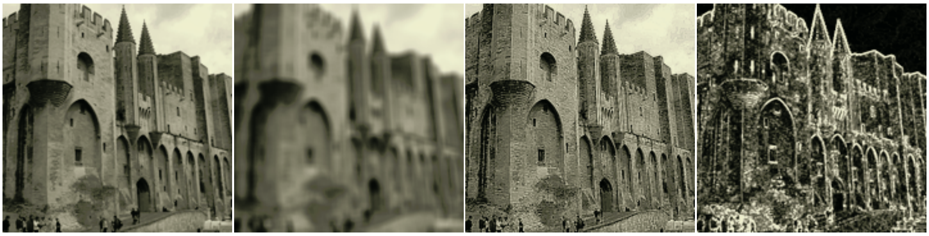

Fig. 1. Examples of image processing convolutions. The original image (left) after blurring, sharpening, and edge detection are performed, respectively *(Image courtesy of Patrick Flynn)*

Convolutions become costly as the size of the filter mask grows. While 3x3 filters are useful in many applications, filter sizes as large as 65x65 and beyond are used, as well. On a 2048x2048 image, a total of 65x65x2048x2048 (over 17 billion) array lookups, multiplications and additions are needed to perform the straightforward convolution operation. Caching and other concerns quickly drive up the cost of this computation. Fortunately, it is possible to optimize the convolution operation by making use of Fourier processing, which will be discussed in further detail in the next section.

3 Implementation

The first step in comparing the GPU to the CPU in performing digital image processing operations will be to perform simple averaging filter convolutions across images ranging in size from 64x64 to 2048x2048. This size range includes the small image sizes common in texture maps developed for 3D games as well as the large image sizes that include full-screen views on the best available current display technology and consumer-level high-resolution digital camera images. In previous work [8], 3x3 filters were shown to have a significant speed advantage on the GPU over the naïve CPU implementation. Two new items remained to be addressed: how the GPU's advantage scales for larger filter sizes, and how the GPU compares to FFT-optimized CPU code.

3.1 DIP on GPU

Using Cg, programming a convolution filter on the GPU is a very straightforward process. Figure 2 shows a section of code for a simple 3x3 averaging filter in Cg. A texture lookup function (`texRECT`) makes it easy to find neighboring pixel values relative to the output pixel's coordinates (`texCoord`) by adding a vector (`float2`) to describe the relative position (up one pixel, left one pixel, etc.). Performing more advanced convolutions, such as edge detection, Gaussian smoothing filters, embossing, and more, only requires adding a multiplier to each texture lookup corresponding to the weight of the mask for that pixel's position.

The C version of a convolution, as shown in Figure 3, is more difficult for several reasons. First, the C version must treat each color component (red, green, and blue, or *RGB*) as separate computations, while the GPU computes all three components simultaneously as three-component vectors of type `float3`. Second, the C method computes array positions in a one-dimensional manner corresponding to linear addresses in main memory rather than the 2D relative indexing of Cg. Therefore, non-intuitive modular arithmetic is necessary to resolve pixel values. Furthermore, the Cg version does not depend upon the size of the image, due to the fact that the program is written to operate on a per-pixel basis. The C version must know the width and height of the image in order to perform the needed array index lookups.

```
float3 a[9];

a[0] = texRECT(image, texCoord + float2(-1, 1));

a[1] = texRECT(image, texCoord + float2(0 , 1));

…

a[7] = texRECT(image, texCoord + float2(0 ,-1));

a[8] = texRECT(image, texCoord + float2(1 ,-1));

color = (a[0]+a[1]+a[2]+a[3]+a[4]+a[5]+a[6]+a[7]+a[8])/9.0;
```

Fig. 2. A 3x3 averaging filter in Cg

```
int a1,a2,a3,a4,a5,a6,a7,a8,a9;
// wd and ht are the width and height dimensions of the RGB image im1
int m = wd*ht*3;
for (r=0; r<m;r++)
{
  a1=(r-(wd+1)*3+m)%m;
  a2=(r-(wd)*3+m)%m;
  …
  a8=(r+(wd)*3)%m;
  a9=(r+(wd+1)*3)%m;
  im2[r] = (im1[a1]+im1[a2]+im1[a3]+im1[a4]+im1[a5]+im1[a6]+im1[a7]+im1[a8]+
            im1[a9])/9;
}
```

Fig. 3. A simple subroutine in C for computing the 3x3 Averaging Filter

3.2 GPU Versus FFTW

Performing a fair comparison of the power of the GPU over the CPU in performing 2D convolutions involves more than the straightforward convolution. On the CPU, it is significantly faster in most cases to convert the image from the spatial domain to the frequency domain by means of the Fast Fourier Transform (FFT), then multiply the frequency representation of the image by the frequency domain version of the convolution filter, then take the inverse FFT, or IFFT, to acquire the filtered image.

$$F(u,v) = \frac{1}{N^2}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y)[\cos(2\pi(ux+vy)) - i\sin(2\pi(ux+vy))] \quad (1)$$

The Discrete Fourier Transform (DFT) of an NxN image whose pixel values are stored in a matrix *f(x,y)* is described by the formula given in Equation 1 above. The Fourier Transform converts a 2D image in the spatial domain into a 2D matrix of frequency values. This frequency representation of the image is useful due to the fact that a spatial convolution of a filter mask of size MxM over an image of size NxN translates to a multiplication of the frequency domain representations of the two matrices. The FFT is simply a time-optimized approach for computing the DFT of an image.

A spatial 2D convolution of an image matrix *f* of dimension *NxN* and a filter mask *h* of size *MxM* is performed according to the formula given in Equation 2 below. Notice that every element of the image matrix is multiplied by every element of the mask matrix, for $O(N^2M^2)$ operations. For large filter mask sizes, this approaches $O(N^4)$ time. The FFT makes it possible to obtain the frequency domain representation of an image in $O(N^2log^2N)$. Assuming a pre-computed FFT of the filter mask, the element-wise multiplication of the image and mask matrices in the frequency domain takes only N^2 multiplications, followed by the IFFT, which involves the same computation time as the FFT, for a total of $O(N^2log^2N)$ time, which can be significantly faster than $O(N^4)$ for realistic values of N.

$$g(x,y) = \sum_{s=-w}^{w} \sum_{t=-w}^{w} h(s,t) f(x+s, y+t), \text{ where } w = \frac{M-1}{2} \tag{2}$$

In most image manipulation and processing packages, Fourier-domain processing is used to perform convolutions expressly because of the acceleration factor. For example, in a first trial run using the naïve convolution of a 15x15 mask on a 1024x1024 image took 6.5 seconds, while the FFT version took 0.54 seconds to perform the same convolution using the frequency-domain approach, a speedup of more than 12:1.

The steps necessary to convolve a filter over an image in the frequency domain are as follows. First, the mask is padded with zeros to match the size of the input image, and the FFT of the mask is computed. Because this only needs to be done once for each filter and size, this is usually precomputed and does not appear in the processing times given in the results. Second, the input image is converted to its frequency representation by performing the FFT. Note neither the mask nor the image should be shifted to the center, as is frequently done in Fourier analysis (if the filter mask is placed at the center of the zero-padded matrix, the input image must be shifted by multiplying every odd element by –1 before the FFT is performed). Third, an element-wise multiplication of the two matrices is performed. Fourth, the inverse FFT is performed on the resulting matrix. Fifth, the complex components of the IFFT are dropped, and the real values now contain the result of the image convolution. A highly optimized FFT package [3] will be used to test the speedup of this approach.

4 Results and Discussion

Straightforward algorithms for convolutions on the GPU and on the CPU were developed for comparison across a wide variety of matrix and image sizes (64x64 to 2048x2048). The GPU proved to be much faster than the CPU at straightforward convolutions like Gaussian smoothing, averaging filters, and edge detection for filter masks from 3x3 to 29x29. Optimized CPU code was developed using the FFTW library, version 3.0.1 [3]. In the case of a 2D 3x3 convolution filter, the GPU (an NVIDIA GeForce FX6800) was up to 130 times faster at high resolutions (2048x2048) than the CPU (a dual 2.8 GHz Intel Pentium 4). The GPU was from four to 139 times faster than the FFTW-optimized version at performing a 3x3 convolution across images from 64x64 to 2048x2048. Figure 4 shows a logarithmic scale comparison of the GPU and FFTW versions over the naïve CPU version. Note that at 2048x2048 the CPU versions process less than one frame per second (fps), while the GPU processes over 60 fps, sufficient for real-time interactivity. Frame rate values are shown for the GPU and FFTW versions; naïve CPU numbers are given in Figure 5.

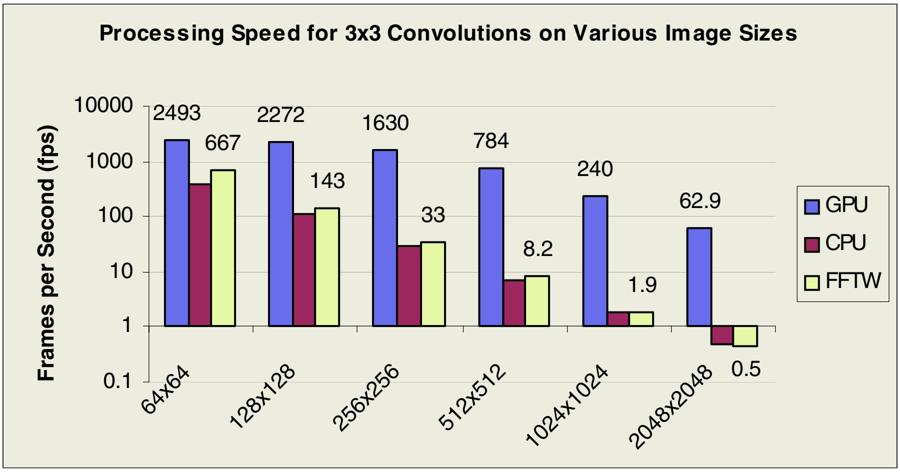

Fig. 4. Logarithmic scale speed comparison of convolving a 3x3 filter mask over images of varying sizes on GPU vs. naïve CPU convolution and FFTW-optimized Fourier processing

The GPU is especially well-suited to performing 2D convolutions and morphological masking and filtering operations. Furthermore, programming the GPU version of these algorithms is a straightforward process, allowing the developer to access pixel neighborhoods using a relative indexing paradigm rather than a complicated modular arithmetic scheme for referencing 2D array elements in main memory. The GPU provides the greatest speed advantage at smaller convolution kernel sizes in the 3x3 to 29x29 range, where the FFT provides the least advantage over the CPU. The GPU could therefore provide significant acceleration in applications where filter masks are small (edge detection, etc.) and where a compatible GPU is available.

Figure 5 shows the results of applying averaging filters of sizes ranging from 3x3 to 29x29 on images from 64x64 to 2048x2048. The table entries show the results for the GPU, followed by the naïve CPU convolution, in each cell. The bottom row shows the time for the FFTW-optimized filters. The FFTW version is given once for each image size due to the fact that the FFT approach to convolution does not vary in processing time by filter size, because each filter must be padded with zeros to match the size of the input matrix before Fourier processing in order to multiply by the image matrix in the frequency domain. Therefore, it is just as fast to process an image with a 65x65 filter as it is to use a 3x3 filter when using Fourier domain acceleration.

Table 1. Timing results for the convolution of various filters (3x3 to 29x29) over various image sizes (64x64 to 2048x2048) on GPU/CPU (in frames per second)

Frame rates for applying convolution masks over various input image sizes (in fps)						
	64x64	128x128	256x256	512x512	1024x1024	2048x2048
3x3	2493 / 388	2272 / 111	1630 / 29.7	784 / 7.27	240 / 1.9	62.9 / 0.48
5x5	2293 / 299	1877 / 85.8	1019 / 22.3	358 / 5.56	96.7 / 1.4	23.6 / 0.35
7x7	2008 / 169	1433 / 45.6	634 / 11.5	193 / 2.93	50.1 / 0.73	12.3 / 0.18
9x9	1739 / 108	1099 / 28.2	422 / 7.21	120 / 1.79	30.3 / 0.45	7.2 / 0.11
15x15	1077 / 35.9	534 / 9.92	166 / 2.52	44.3 / 0.62	11 / 0.15	2.59 / 0.038
17x17	929 / 30.5	437 / 7.73	131 / 1.95	34.1 / 0.48	9.6 / 0.12	2.06 / 0.029
19x19	796 / 24.3	360 / 6.18	100 / 1.55	27.5 / 0.38	6.8 / 0.095	1.65 / 0.024
29x29	415 / 10	168 / 2.53	46.6 / 0.63	11.9 / 0.16	2.98 / 0.039	0.72 / 0.009
FFT	*667*	*143*	*33*	*8.2*	*1.9*	*0.45*

Figure 6 shows the speedup derived from dividing the GPU frame rates by the naïve and FFTW-optimized CPU frame rates. Note that the GPU is from 6 to 132 times faster than the straightforward CPU convolution, with an average speedup over these values of 59:1. The GPU is faster than even the FFTW implementation by an average of 18:1 for these filter sizes. At 29x29, however, the FFTW version begins to match the GPU in speed. Interestingly, this is also the largest 2D convolution filter we were able to implement in Cg. The code for a mask at this size must perform 29x29, or 841 texture lookups, multiplications, and additions per pixel, or around 2500 pixel operations, with 3 components per pixel (*RGB*). While the specifications for the 6800 series GPU state a maximum operation count of 65,536, we were unable to successfully implement filters any larger than 29x29.

Table 2. Speedup achieved in Figure 5. Figures are for GPU over naïve/FFTW-optimized CPU versions, rounded to the nearest whole number

Speedup of the GPU over the naïve CPU / FFTW convolutions						
	64x64	128x128	256x256	512x512	1024x1024	2048x2048
3x3	6 / 4	20 / 16	55 / 49	108 / 96	126 / 130	132 / 139
5x5	8 / 3	22 / 13	46 / 31	64 / 44	69 / 52	68 / 52
7x7	12 / 3	31 / 10	55 / 19	66 / 24	69 / 27	68 / 27
9x9	16 / 3	39 / 8	59 / 13	67 / 15	68 / 16	66 / 16
15x15	30 / 2	54 / 4	66 / 5	71 / 5	71 / 6	68 / 6
17x17	30 / 1	57 / 3	67 / 4	70 / 4	80 / 5	71 / 5
19x19	33 / 1	58 / 3	65 / 3	71 / 3	71 / 4	69 / 4
29x29	42 / 1	66 / 1	74 / 1	76 / 1	76 / 2	73 / 2

It was mentioned that it was assumed that the FFT of each filter had been precomputed for timing purposes. If a new filter is used on each image, a processing penalty of 30% or more will result from adding another FFT to accommodate the new filters. Another important distinction between the convolution on the GPU and the FFT-based approach is that the FFT produces a cyclical convolution when used as described above. For visualization purposes, this is usually close enough to a linear convolution to produce acceptable results. If, however, a true linear convolution is required, the FFT routine must pad the image with zeros to increase the image size to twice its original height and width, causing a four-fold decrease in speed.

One other hidden consideration exists when performing the FFTW-optimized version of these operations. The first time the FFTW package is run on a machine for a given image size, the library creates an optimization strategy specific to the machine's architecture, memory, and other attributes. For small matrix sizes, this planning step took only a second or two, but for the 2048x2048 case, it took over 3 ½ minutes. Fortunately, the FFTW library provides a mechanism for saving and recovering the optimal plan so that this process need only be performed once, but the time consideration can be significant for the first use.

5 Conclusions and Future Work

In convolution-based operations from digital image processing (DIP), the GPU showed an average speedup of 59:1 over the straightforward CPU convolution and 18:1 over the FFTW-optimized approach for filter sizes from 3x3 to 29x29 on images from 64x64 to 2048x2048. Notably, the 60 fps speed of performing 3x3 convolutions on 2048x2048 images on the GPU could enable better and cheaper real-time high-resolution applications for scene reconstruction, navigation, and security purposes. Besides speed-up, GPUs have a programming advantage in Cg over CPUs programmed in C. This is due to the fact that pixels can be referenced in Cg in a relative fashion, such as one pixel to the right and down from the current location, unlike the more complicated 1D memory array lookup calculations needed in C.

Because 2D DIP is well-suited to GPU acceleration, a fruitful area for future research would be developing an image processing API or adding GPU acceleration to an existing image processing package. An API or multi-step compiler system like Brook [1] designed specifically for DIP operations could be a significant step toward general-purpose utilization of the GPU as a parallel vector processor.

References

1. Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P.: Brook for GPUs: Stream Computing on Graphics Hardware, *Proceedings of ACM SIGGRAPH 2004.* ACM Press, (2004) 777-786.
2. Dalvi, A. Intel targets 4 GHz barrier. *Techtree*, online news service. Available online at http://www.techtree.com/techtree/jsp/showstory.jsp?storyid=3970. (2003).
3. Frigo, M., Johnson, S.G.: FFTW: An Adaptive Software Architecture for the FFT, *IEEE ICASSP Proceedings (3).* IEEE Press, (1998) 1700-1703.

4. Jain, A.K.: *Fundamentals of Digital Image Processing,* Prentice Hall, Englewood Cliffs, NJ, (1989).
5. Mark, W.R., Glanville, R.S., Akeley, K., Kilgard, M.J.: Cg: a system for programming graphics hardware in a C-like language, *ACM Transactions on Graphics.* ACM Press, (2003) 896-907.
6. Moreland, K., Angel, E.: The FFT on a GPU, *Proceedings of the ACM SIGGRAPH Conference on Graphics Hardware*. Eurographics Association, (2003) 112-119.
7. NVIDIA Corporation: GeForce 6800 Product Overview. http://www.nvidia.com (2004).
8. Payne, B.R.: Accelerating Scientific Computation in Bioinformatics by Using Graphics Processing Units as Parallel Vector Processors. (Doctoral dissertation, Georgia State University, 2004). *Dissertation Abstracts International* (UMI. No. pending).