J Real-Time Image Proc
DOI 10.1007/s11554-008-0085-x

ORIGINAL RESEARCH PAPER

MinGPU: a minimum GPU library for computer vision

Pavel Babenko · Mubarak Shah

Received: 3 September 2007 / Accepted: 30 April 2008

Abstract In the field of computer vision, it is becoming increasingly popular to implement algorithms, in sections or in their entirety, on a graphics processing unit (GPU). This is due to the superior speed GPUs offer compared to CPUs. In this paper, we present a GPU library, MinGPU, which contains all of the necessary functions to convert an existing CPU code to GPU. We have created GPU implementations of several well known computer vision algorithms, including the homography transformation between two 3D views. We provide timing charts and show that our MinGPU implementation of homography transformations performs approximately 600 times faster than its C++ CPU implementation.

Keywords GPU · GPGPU · Computer vision · Homography transformation

1 Introduction

Little research was conducted on general-purpose computation on GPU before 2000. Since 2002, there has been a great deal of interest in GPU. A comprehensive list of articles is available at [5]. In 2001, Rumpf and Strzodka [24] described a method to implement level sets on GPU. In 2002, Yang and Welch [27] studied image segmentation and smoothing on graphics hardware. There were several articles published in 2003–2004, many of which appeared in computer graphics conferences rather than computer vision. For example, in 2003 Moreland and Angel [18] published an article in SIGGGRAPH conference on how to implement fast Fourier transform on GPU. Later that year, Colantoni et al. [4] described color space conversions, PCA and diffusion filtering on GPU. In 2004, 'GPU Gems 2' dedicated a chapter to computer vision on GPU [8]. It described correcting radial distortions, Canny edge detector, tracking hands, and image panoramas on GPU. Yang and Pollefeys [26] published a CVPR paper later that year, on implementation of real-time stereo on GPU. Ohmer et al. [22] implemented a face recognition system using Kernel PCA and SVM. Labatut et al. [16] utilized level set-based multi-view stereo. The well-known SIFT algorithm was employed in 2007 by Heymann et al. [14]. In 2006, another implementation of Fast Fourier transform was done by Govindaraju et al. [11]. Other methods that were implemented include [5]: generalized Hough transform, skeletons, depth estimation in stereo, motion estimation, 3D convolutions, contour tracking, segmentation and feature tracking, and matching. However, because articles before 2005 used previous generations of graphics hardware, their contribution is often outdated.

MinGPU is a library which hides all graphics-related routines from the end user, which is particularly useful for those who do not have particular skills or desire to delve into low-level details of GPU programming. MinGPU can be used as a basic building block for any GPGPU application. It can help implement any algorithm on GPU. Besides MinGPU, there exists another library which is dedicated to computer vision on GPU. OpenVIDIA [9, 21] is an open source library which contains many useful vision algorithms, like edge and corner detection, feature-based corner, object tracking, and image enhancement/preprocessing routines. The complete algorithm list can be found at [21]. OpenVIDIA also maintains a web page

P. Babenko (✉) · M. Shah
Computer Vision Lab,
School of Electrical Engineering and Computer Science,
University of Central Florida, Orlando, FL, USA
e-mail: pavelb@cs.ucf.edu

which lists many recent computer vision related papers [5]. However, this library is not built upon a reusable core. Another notable effort is the BrookGPU library [2]. BrookGPU was conceived as an extension to the C language which supports stream programming and data-parallel constructs. It includes a compiler and run-time support libraries. This library has not been updated since 2004. The latest v0.4 release of BrookGPU can be downloaded from [3]. Yet another open source GPU library is Sh [25]. Sh is a GPU metaprogramming library designed in C++. It defines GPU objects and data types as C++ classes, and also defines operations on these objects in terms of C++. Control structures are defined as C++ macros. Thus, it is possible to write programs for GPU using C++ notation. However, the target users for this library are graphics developers; this library serves as a substitute for GPU languages like Cg, GLSL, or HLSL. Both these medium-sized open-source projects feature little documentation and are complex for users, who do not have advanced knowledge of graphics processors and C++ programming. RapidMind [23] is a library similar to Sh in that it wraps GPU constructs into C++ classes. In contrast to Sh, the target users of RapidMind are multi-core software developers. RapidMind is an attempt to provide a unified high-level development platform for parallel computations; it works with Cell, multi-core CPUs and GPUs. This commercial library is also somewhat advanced in use and does not give a direct control over GPU processors.

Our work makes two contributions to computer vision. First, we have created a C++ class library, MinGPU. This library can function as an introduction to GPU world for researchers, who have never used the GPU before. We intentionally designed the library and interfaces to be as straightforward as possible. MinGPU provides simple interfaces, which can be used to load a 2D array into the GPU and perform operations. All GPU and OpenGL related code is encapsulated in the library; therefore users of this library do not need any prior knowledge of GPU. The library is configurable for use in a variety of purposes. It is available online on our web site at [17]. Our MinGPU library is designed to work on nVidia 7000 series cards as well as on ATI Radeon cards series R400 (ATI X800) and later versions. We have designed and tested our library on nVidia 7300LE card, which is a basic card in 7000 series and is in widespread use. Some portions may be functional on nVidia 6000 series cards as well as on former ATI cards.

In performance evaluations we used, unless stated otherwise, nVidia GeForce 7300LE graphics card ('GPU') installed in DELL XPS 410 desktop, featuring Core 2 Duo 2.4 MHz processor and 2 Gb of memory ('CPU').

Second, we have implemented three popular computer vision methods on GPU, as well as the number of simple image operations like taking image derivatives, Gaussian smoothing and image pyramids. In this paper, we have included implementations of pyramids and homography transformations on GPU, while the descriptions and listings of all other methods, including Lukas–Kanade optical flow and optimized normalized cross-correlation, are included in an extended technical report which can be downloaded from our site [17]. For each of these methods, we present timing and provide a detailed timing comparison of GPU homography transformation versus its non-GPU implementations.

GPU programming using OpenGL is not a trivial matter even for experienced programmers. In summary, MinGPU is a small, simple library which provides a quick introduction to a programming on GPU. It works on most graphics cards currently installed in personal computers, making image processing algorithms portable between different graphics devices from ATI and nVidia. Most PC users probably don't realize that their PC is equipped with a parallel processor, which they can use with the help of libraries like MinGPU.

This paper is organized as follows: in Sect. 2, we present general information on the current state of art and the evolution of GPUs, their architecture and design. In Sect. 3, we give a brief introduction of our C++ class library for the GPU-MinGPU. In Sect. 3 we also show how to implement few common computer vision algorithms with MinGPU, image derivatives and image pyramids. Sect. 4 contains the MinGPU implementation of a more complex computer vision method—the homography transformation between two 3D views, as well as speed comparisons and discussions. The MinGPU class structure can be found in Appendix 1. The Appendix 2 contains code listings of all programs mentioned in Sects. 2, 3, 4.

2 Current trends in GPU

2.1 Graphics processors

Most of today's graphics cards from the largest GPU makers, nVidia, and ATI, contain two processors, the vertex processor and the fragment processor.

2.1.1 Vertex processor

All graphics cards operate on 3D polygons. Every polygon is defined by (x, y, z) coordinates of its vertices. For example, every triangle is defined by three vertices. The camera vertex is usually set to be at (0, 0, 0). When the camera moves, the coordinates of all polygon points must be re-computed to reflect a change in the camera position. This operation is performed by the vertex processor. Vertex processors are specially designed to perform this operation

and therefore, are able to optimize the speed of coordinate transformations. After the coordinates are recomputed, the vertex processor determines which polygons are visible from the current viewpoint.

2.1.2 *Fragment processor*

After the vertex processor re-computes all the vertex coordinates, the fragment processor covers the visible portions of the polygons with textures. The fragment processor does this with a help of the 'shader' programs. Originally, in the computer graphics world, the purpose of the 'shader' program was to add graphic effects to textures like shade (hence comes the name), but now this feature is being inherited by general-purpose GPU users. Up until a few years ago, all shader programs were written in the assembly language. However, as graphics hardware evolved and became capable of executing much larger shader programs a need for a specially designed language became evident. Many contemporary shader programs are C-like programs written in Cg, 'C for graphics' language. Cg language was created by nVidia. nVidia supplies manuals and examples on Cg, which can be found in the Cg Toolkit [7].

The most important difference between contemporary CPUs and GPUs is that GPUs run programs concurrently and are SIMD-type processors. The programs are executed for every output texture pixel independently. Therefore, if the GPU has 8 instruction pipelines in the fragment processor, it is able to execute the same program on up to 8 texture pixels simultaneously. Contemporary fragment processors have 4–128 instruction pipelines.

While both the vertex and fragment processors are programmable, we are more interested in the fragment processor because it is specifically designed to work with textures which, in our case, can be viewed as 2D data arrays. Therefore, all algorithms in this paper are designed to work on a fragment processor. On the other hand, vertex processor is optimized to work with pixels.

At the time of writing, the typical upscale desktop computer is equipped with Intel Core 2 Duo processor working at 2.4 GHz. Let's roughly compare a productivity of this processor to a productivity of nVidia 7300LE (light edition) GPU, which is a commonplace graphics card installed in the same desktop. We assume single-core CPUs and no SIMD CPU operations are used. The clock rate of 7000 series nVidia GPUs lies in 400–700 MHz range. The 7300LE runs at 450 MHz. There are 4–24 pipelines in fragment processor in the currently popular 7000 series. The 7300LE contains 4 pipelines. We also take into account that nVidia GPU pipelines can typically process two pixels in one clock cycle and they process each of the pixel's 4 color channels simultaneously. Each pixel is represented as a 4-float number (RGBA). Therefore, if we set up our array so that each of RGBA float assumes a data value (see Sect. 2), we gain an additional four times the speed. After we multiply all the increases in speed, nVidia 7300LE works as a processor with a virtual 14.4 Ghz rate, due to parallelism; this rate is already six times higher than that of latest Intel Core 2 Duo 2.4 GHz CPU. The Table 1 illustrates trends in GPU developments in recent years.

Consequently, we find that the modest GPU card installed in our computer has a performance about six times higher than the latest CPU. If we install the latest nVidia 8800 GPU (1.5 GHz, 128 pipelines) it could yield a theoretical speedup of 106 times compared with the state-of-the-art CPU.

It is hard to make an exact performance comparison of current CPUs to GPUs for many reasons. Modern processors are also equipped with technology which allows simultaneous processing of multiple data by the same instruction. Starting with Pentium II, all Intel processors are equipped with the MMX technology which allows sequential processing of 4 integers. All Intel and AMD processors newer than Pentium III include SSE—similar technology with processing ability of 2 to 4 floating point numbers simultaneously. We also have to mention a recent trend to include more than one processing core on chip. For example, recent Pentium processors feature 2–4 cores which share the same on-chip cache. Every core can be viewed as separate processor. Nevertheless, the productivity of current graphics processors significantly exceeds the productivity of CPUs and this trend will continue.

Table 1 The trends in GPU evolution in recent years

Graphics card	Year	Shader unit clock rate (MHz)	Shader units	Texture fill rate (billion/s)	Memory bandwidth (GB/s)	Memory max (MB)
GeForce 8800 Ultra	May 2007	1500	128	39.2	103.7	768
GeForce 8800 GTX	Nov 2006	1350	128	33.6	86.4	768
GeForce 7900 GTX	Mar 2006	650	24	15.6	51.2	512
GeForce 7800 GT	May 2005	550	16	9.6	32.0	256
GeForce FX 6800	Jun 2004	400	16	6.4	35.2	256
GeForce FX 5800	Jan 2003	400	4	4.0	12.8	256

On the other hand, due to physical constraints it is impossible to run a transistor-based conventional processor at a significantly higher speed than what is already attained by the industry. Theoretically the only way to circumvent this limit lies in the use of different base technologies, like quantum technology. At the time of writing such technology was many years down the line. Researchers in computer vision are aware that many vision algorithms are not currently able to run in real-time due to their high computational complexity. We feel that the one way to make this possible in observable future is the use of parallel technology, such as those present in graphics processors.

2.2 GPU limitations

All GPUs suffer from two drastic design limitations. The first limitation is, ironically, the fact that GPU computations are done in parallel. The algorithm must be ready to work in multi-thread mode in order to operate on the GPU, which is not a feature of many algorithms. In the GPU, every pixel in the 2D texture is processed independently. It may not be possible to know the sequence in which pixels are processed; therefore it is not possible to pass any data between pixels while they are being processed. For example, let's consider a popular connected components algorithm which is used in many vision algorithms today, the Canny edge detector. There exist two versions of a connected components algorithm: recursive and non-recursive. A recursive version cannot be implemented on the parallel processor, because being "recursive" implies the knowledge of the order in which pixels are being processed which is not present in a parallel processor. A non-recursive version of the connected components algorithm uses a running variable, which contains the largest region label currently assigned. It is not possible, as it was stated above, to maintain this counter on the parallel processor. There exist some parallel versions of connected components [13]; however, those versions use binary structures like graphs and trees, which are hard to implement on the GPU. Currently, we do not know of any successful implementation. In the GPU, when one processes a pixel at (x, y) location one can only write the result into (x, y) location of the output texture. There may be more than one output texture (as many as 4–24 for 7000 series and up to 48 in 8000 series). We must also consider that every pixel is comprised of 4 floats (RGBA value). For the 7300LE card, one can write 16 values for every pixel processed, but they cannot be written into any other than (x, y) location in the output textures. This is the second limitation of graphics hardware. Only the latest CUDA-technology based graphics cards from nVidia allow scatter operations in addition to gather operations.

For example, let's consider building a color histogram of a grayscale image. Every pixel in the input image may take values in the range of 0–255, which means there are 256 bins in the color histogram. For every pixel processed, we must increment one of 256 bins, which due to the above limitation is not possible on the GPU. This is a very simple algorithm, yet it is not possible to implement it on the GPU. One source [21] devised an approach which computes approximate color histogram on small set of input values.

Many other researchers agree that the most promising algorithms to implement on the GPU are filter-like algorithms, which process every pixel independently of the others. Examples of this include Gaussian smoothing, convolutions, image pyramids, geometric transformations, image de-noising, cross-correlation as well as many other algorithms.

Contemporary computer architecture features one impasse with respect to the GPU which we cannot avoid mentioning, which is the transfer rate between main (CPU) memory and GPU memory. The latest computers are equipped with PCI Express memory bus, which is the fastest expansion bus to date in desktop computers. This memory bus has a full duplex transfer rate of 250 MB/s for every lane (500 MB/s for PCI Express 2.0 released in January 2007). There may be up to 32 serial lines, however many commodity computers are equipped with less than that. We measured main memory to GPU memory bus transfer rate on our new DELL XPS 410 desktop to be approximately 860 MB/s. At this speed it would take approximately 4–5 ms to transfer an array of 1 million 32-bit floating point numbers (1 k $\times$ 1 k image) from the CPU to GPU. Simple operations (addition, for example) over the same array in the CPU (Core 2 Duo 2.4 MHz) would take about 4 ms. Therefore, the time required to complete simple array operations time such as transfering an array from CPU to GPU is comparable to the time required for applying an operation in the CPU. An example of this backlog is given in the beginning of Sect. 3; we would like to point out that some older GPU cards feature slower GPU to CPU transfer rate than CPU to GPU. An interesting recent trend in computer design is the appearance of expansion connectors for HyperTransport buses, the front-side buses used in many today's computers as a fast link between processor and a main memory. The HyperTransport protocol supports 41.6 GB/s bandwidth in two directions making it much faster than PCI Express. Recently, plug-in cards, such as fast speed coprocessors, have appeared and can access the HyperTransport bus.

3 MinGPU library

In the area of computer vision, we often encounter situations in which we need to process every point of a 2D

array (for example, in an image). It is done in a double for loop as shown in the 'Hello, World!' example in Sect. 3.1. Any 2D array can be represented as a GPU texture. If we do not need to carry over any information between the points of this array, we can implement the inner part of this double loop as a shader (Cg) program. The array is uploaded into the GPU as texture. Then, the shader program is run and the output texture downloaded back into main memory.

In this section we introduce the smallest possible library, MinGPU, which can implement the above mentioned code on a GPU. We attempted to convert the CPU code into GPU in a straightforward manner. In Sect. 3.1, we present an implementation of this double loop in MinGPU. The rest of the section is dedicated to few more MinGPU examples based on simple vision algorithms.

This section uses a learn-by-example technique. We progress from simple, for instance taking image derivatives and computing image pyramids to more elaborate examples, such as an implementation of a homography transformation on GPU. In the following section we present an elaborate GPU example, an implementation of a homography transformation on GPU.

MinGPU is a C++ class library. Due to the incapsulation paradigm, users do not need any knowledge of its inner structure; therefore they do not need any knowledge of the details of how the fragment processor or OpenGL drivers operate. MinGPU contains only two classes: *Array* and *Program*. The *Array* Class defines a 2D array in the GPU memory, while the *Program* class defines a Cg program in the GPU memory. All class methods are listed in Appendix A. In a straightforward scenario, the user prepares the data array and uploads it to the GPU using the methods from the *Array* class. The Cg program is then loaded and compiled. Program parameters can be set using the method from *Program* class. The Cg program is then run and the results are generated in the GPU memory, which are then downloaded to CPU memory by another call to a method from the *Array* class. The 'Hello, World' example illustrates this.

3.1 'Hello, World!' example

We are going to convert this simple CPU code into GPU:

```
for (row = 0; row < MaxRow; row ++)
{
  for (col = 0; col < MaxCol; col ++)
  {
    Array[row][col] ++;
  }
}
```

All code listings are given in the Appendix 2. The code which implements 'Hello, World' on MinGPU is given in Listing 1. This, as well as the most other listings, contains two pieces: a C++ program and a Cg program. Let's first look at C++ program. It matches the idea we discussed above in a straightforward way: we create both array and Cg programs, copy the array to a GPU, set program parameters and run it.

As for Cg program, we won't be able to cover the entire Cg language here (which you can find in a Cg user manual [6]), but we can highlight some key points. First, our original program contains just one function, *main*, which is its entry point. There are two input parameters to this function, a parameter *coords* of type *float2* and a parameter *texture* of type *samplerRECT*. The parameter *coords* is bound to pre-defined name *TEXCOORD0*, which contains (x, y) coordinates of the currently processed pixel. Pre-defined names are set automatically by the hardware, so we do not need to provide values for these. Subsequently, the parameter *texture* is a regular parameter which we must initialize with a call to the *SetParameter* function. It is our input array, which we want to increment by 1. The standard Cg function *texRECT* fetches a value of *texture* array at *coords* coordinates. In this simplified example, we used the same Cg texture as both an input and an output array. We store the intermediate value in the *result* variable, a standard 4-float RGBA type. The assignment string $result = y + 1$ increments each of four float values in *result* by 1. In this way, every float in *result* will contain the same value.

Cg functions return one 4-float RGBA value which is the generated value for the current (x, y) texture pixel. When we download this value to the CPU in luminance mode (which is discussed in the following subsection), OpenGL drivers take a mean of the 4 RGBA floats to produce a single luminance value for each texture pixel. In color mode, all 4 RGBA channels are returned. All Cg programs run on every point in an output, not an input, array. These two arrays can have different sizes.

3.2 MinGPU operating modes

The 'Hello, World' code has some implicit features which we need to clarify. First, we always assume that MinGPU input arrays contain grayscale values in the 0–255 integer range. This array format is common in computer vision. Second, all numbers a GPU operates on are floats of 8–32 bit length. Therefore, MinGPU converts inputted integer arrays into floats. MinGPU uses 32-bit (4-byte) long floats. This is the largest floating point format supported in graphics cards today.

A fragment processor encodes every texture pixel as a four float RGBA value, a *quad*. There is one float value for the red, green, and blue colors and one for the alpha channel. All operations on a quad are always performed on all four floats in parallel by hardware. The Cg language conveniently includes a special vector type definition *float4* to define a quad.

While all MinGPU input arrays are invariably greyscale MinGPU supports two color modes for arrays—a luminance and a color mode. The reason for this is that GPU color modes are manufacturer-dependent. Since a luminance mode is expected for the nVidia family of processors, it is guaranteed to work in all nVidia cards. However, for all current ATI cards a color mode is required. nVidia also fully supports a color mode. For list of color modes supported by different cards, see tables in [10].

In a luminance mode, every float in the quad holds the same value of one input array cell. In a color mode, MinGPU replicates every input value four times, so that each float in a quad contains the same value. Luminance and color modes are compatible on the level of a C++ code and on the level of a Cg program. In MinGPU, a color mode is specified on per-texture basis. The luminance mode is the default. Textures can be created in a color mode by setting *bMode* to *enRGBA* in a call to *Array::Create*.

3.3 MinGPU basic examples

In the following listings, we used a reduced notation. For brevity, we implied that all required initialization has already been done and omit array and program initialization code from C++ program. The example of a simplified 'Hello, World!' is given in Listing 2.

3.3.1 Taking image derivatives

Taking image derivatives is arguably the most straightforward computer vision algorithm. Image derivatives can be taken in three different directions – d_x, d_y, and d_t. As a derivative kernel, we use Prewitt, Sobel or Laplacian 3 by 3 gradient operators.

The C++ code and Cg program for taking image derivative in the d_x direction are given in Listing 3. *Texture* contains an input array and *Kernel* contains a smoothing kernel. This code contains one more array, *Output, than the 'Hello World'* example. This array accumulates the derivative results. Initially, the *Output* array does not contain any values, so we do not copy this array from a CPU to the GPU; we create it right in the GPU instead. The array we use in call to *Output.Create* receives the results when we download them from the GPU with the *CopyFromGPU* method.

The programs for completing d_y derivations are the same, except for the kernel. The Cg program for d_t derivations must take two arrays, image 1 and image 2, as input so it must be different. The chosen kernels K for this derivative are a 3×3 matrix filled with a value of one and the number one. We included Cg programs for d_t derivations in Listing 4. Array *T1* is an image at time t and array *T2* is the same image at time $t + 1$.

It must be noted that we cannot use the same texture for both the input and output arrays. This is not possible because the values of the points in the input array are used in calculations of more than one output point. Also, the order of calculations is unknown because all the calculations are done in parallel.

Listing 3 contains the first Cg program with loops. Of all video cards which exist today, only the latest cards from nVidia support loops in hardware; older graphic hardware does not support hardware loops. This is the reason why the Cg compiler unfolds some loops during the compilation. A Cg compiler can only execute trivial loops. Loops with number of iterations dependent on the input parameters cannot be executed. This leads to using a fixed number of iterations in Cg program loops, and consequently multiplies the number of Cg functions. For example, we have to keep a separate *derivative 3 × 3* Cg function for 3×3 derivative kernel, *derivative 5 × 5* Cg function for 5×5 derivative kernel and so on.

3.3.2 Computing image pyramids

The pyramid computation is another simple vision algorithm [1]. Pyramids are useful data structures for representing image regions. The lowest level of the pyramid is the original image. Each subsequent level in the pyramid is one-fourth of the area of the previous level. The process of going from higher to lower resolution is called reduction, while the opposite operation is called expansion. We have shown only the REDUCE operation, which is computed according to the following formula:

$$g_l(i,j) = \sum_{m=-2}^{2} \sum_{n=-2}^{2} w(m,n) g_{l-1}(2i+m, 2j+n).$$

In the above equation, g_l is an image at pyramid level l, matrix w contains the weight mask, and i, and j are the indices for the image's columns and rows, respectively. A C++ code and a Cg program for REDUCE operation are listed in Listing 5.

In this example, the input and output array sizes do not match. Because each pyramid reduction effectively reduces the image size by half in both dimensions, the output array side is half as long as the input array side. The important question is how do we determine the values of array elements outside of the array boundaries? In two previous examples, we were able to access such elements. In all of the aforementioned examples, elements located outside of

the array are assigned the value of the nearest element in the array. This behavior is hard-coded by a call to OpenGL function *glTexParameteri* during array creation. There are no other options; therefore if we need to assign a predefined value, such as 0, to elements lying outside of the array area, we have to fill our array with 0's.

We would like to clarify here what we stated in Sect. 2.2 on the limitations of GPU limits. This section may have given the wrong impression that GPU programming is quite simple. In fact, a majority of algorithms either cannot be implemented on current GPUs or can only be implemented with significant difficulties. The problems arise due to two reasons mentioned in Sect. 2.2. First, the fragment processor does computations for all points in parallel and therefore, the algorithm must be able to work in parallel. In particular, it means that the order in which points are processed is not known. Therefore, there are classes of algorithms which cannot be implemented on GPU, notably all recursive algorithms. Also, any global scope variables, such as counters, cannot exist in a parallel algorithm, only constants can be used. Another inconvenient and often problematic limitation is that the current Cg programs can write only to the location of the element they are currently processing. Quite often algorithms have to be altered to account for that limitation.

We began this section with the 'Hello, World' code, the simplest MinGPU program. Unfortunately, that program does not gain an increase in speed due to the use of the GPU. While execution of the code took 2 ms on our GPU compared to 4 ms on a CPU, there is an additional overhead to transmit the array to and from the GPU, requiring 4 to 7 ms for the 4 MB array. Algorithms which operate according to a 'single array load–single use' scheme will likely not gain a significant increase in speed by uploading to the ordinary GPU. In fact, if the computational portion is small, it can even result in a decrease in speed. To be gain a significant increase in speed from the use of the GPU, an algorithm must operate according to the 'single array load–multiple use' scheme. It is also desirable that the computational portion be large.

In the next section, we demonstrate this thesis by presenting an example of an algorithm which operates according to the 'single load–multiple use' scheme. We demonstrate that the increase in speed, due to uploading the algorithms built upon the aforementioned scheme to a GPU, may be quite significant. Some algorithms which are sluggish on a CPU can run in real-time on a GPU.

4 Further examples of MinGPU implementation

In this section, we present an example of an algorithm which operates according to the 'single array load-multiple use' scheme. This algorithm uses homography transformations. We have implemented this algorithm in Matlab, C++, and MinGPU. We present execution times for both CPU and GPU, and show that the GPU Cg implementation operates approximately 600 times faster than CPU C++ implementation.

MRI imagery provides a 3D snapshot of a human body. Thousands of body slices stacked together gives a snapshot of the inner human. A similar idea has recently been proposed [15] in the area of computer vision, a method which involves fusing multi-view silhouettes to reconstruct the visual hull of the 3D object slice-by-slice in the image-plane. This method approximates an object by a set of planes, slices, parallel to q reference plane. These slices are related by plane homographies in different views. In this section, we present a GPU implementation of this algorithm. The algorithm cannot handle concave objects, however for many other objects, such as human bodies, it works reasonably well. We provide an outline of the idea here, while transformation formulas and detailed explanation can be found in [15].

Images of a scene are taken from different viewpoints, with uncalibrated cameras, with an aim to recover a 3D representation of the objects in the scene. The ground plane homographies between different viewpoints are estimated using SIFT features and RANSAC algorithm. Vanishing points are computed by detecting the vertical lines in a scene and then locating their intersection with RANSAC algorithm. In this way, the geometry of the scene is discovered.

To create a stack of object slices, the fact that warping the silhouettes of an object from the image plane to the plane in the scene using a planar homography is equivalent to projecting a visual hull of the object on the plane is utilized. Foreground silhouettes from different views are warped to a reference view using the homographic transformation. Figure 1c contains an example of what the transformed silhouette of an image may look like. After the transformations are applied, visual hull intersections are found by overlapping transformed silhouettes. This operation is repeated for every slice because each slice resides on different parallel planes; therefore different transformations are required between images and slice planes.

The following two examples illustrate this method. In the first example, we have a single camera and a single object on a rotating pad (Fig. 1a, b). The pad rotates at a constant speed and the camera captures views at equal intervals of time, which gives us views of the same figure from every 6°, resulting in a total of 60 views. We create 100 slices of this figure. For every slice and every view, we apply a homography transformation to the slice plane (Fig. 1c), and then we overlap (or fuse) the transformation results. Results for slices 0, 35, 75, 99 are given in Fig. 1d–g.

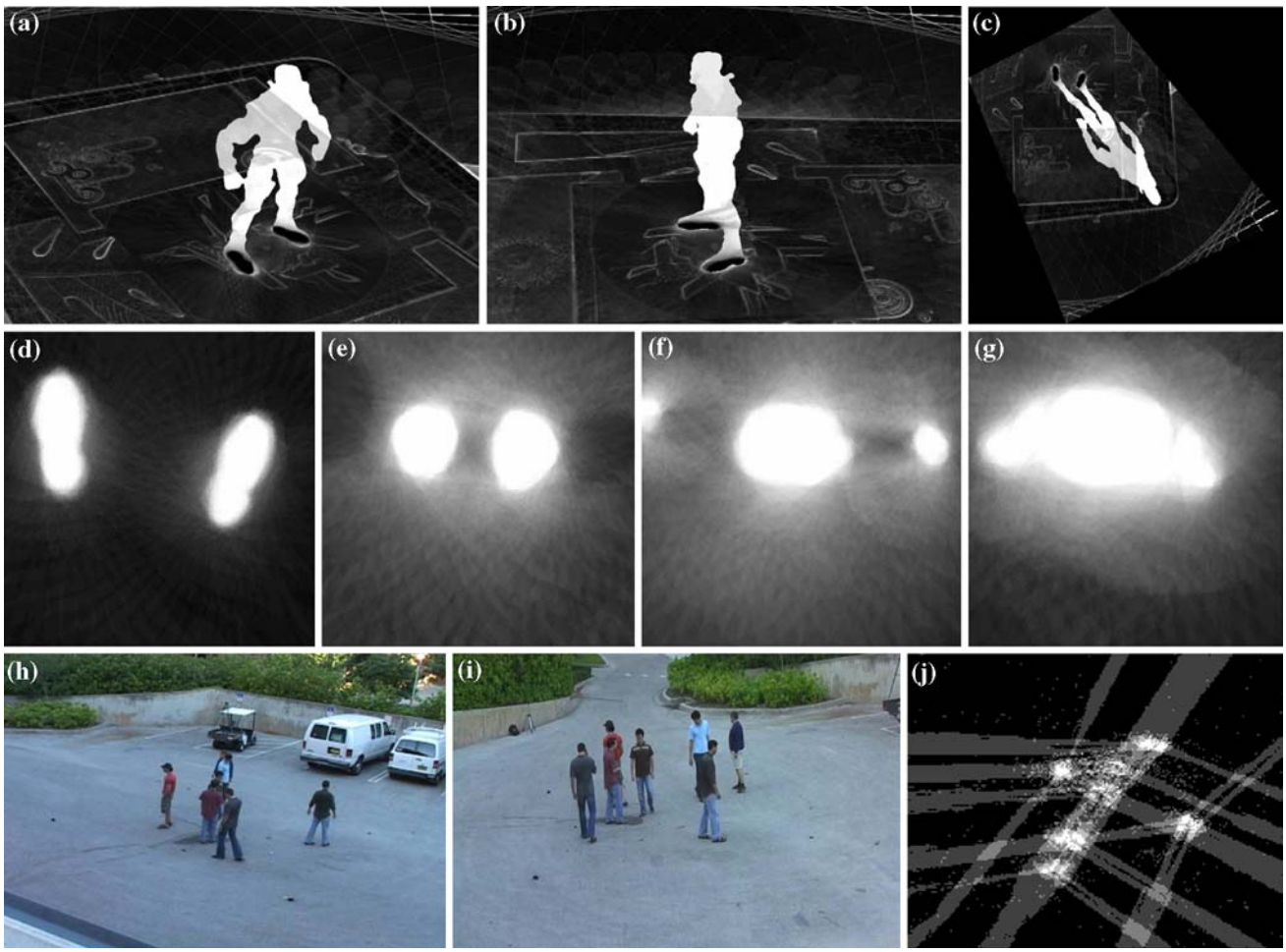

Fig. 1 The results of implementation of visual hull intersection method for reconstructing object model using plane homographies implemented on GPU. **a**, **b** Solid figure on a rotating pad, **c** the homography transformation, **d–g** resulting slices: feet, legs, lower torso, upper torso, **h, i** two views, **j** slice of *h* and *i* at feet level

The second example contains a view of an outside scene (Fig. 1h, i). There are four cameras in the scene, and their relative positions are unknown. People are present in the scene and we have captured pictures of them from all four cameras. We find the scene geometry using the method described above. Then, by applying the homography transformations, we construct a 3D sliced representation of the people (Fig. 1j). The results are coarser than ones in the first example, because here we are using just four views compared to 60 in the first example. The more views we have, the better the outline of the resulting slice.

We input multiple views of the same scene plus all required planar transformation matrices (of 3 by 3 size) into our algorithm. All input views contain binary values, 0 for background pixels and 1 for foreground pixels. The desired number of slices is 100. For every slice, each view will have a separate transformation matrix to a ground plane.

We preload all views into the GPU. Then, for every slice, we apply a homography transformation to every view and integrate the results in the accumulator array. After all views are processed the accumulator array contains the 'sliced' image (Fig. 1d–g, j), which is transferred back to the CPU. Then, the accumulator array is zeroed and we proceed to the next slice. Therefore, this method clearly belongs to the 'single image load–multiple use' category.

When we apply the homography transformation to a view, the resulting image becomes approximately ten times larger than the original view and the points become too sparse. To overcome this problem, we have chosen to use inverse transformations. For every point in the resulting image, we apply an inverse transformation and find its value in the original view through bilinear interpolation. We also crop and rescale the resulting image. We take a region that is 200 × 180 points which are centered at the coordinate origin. When filling this region, we consider every 5th point in both the x and y directions. The C++ code and Cg program are included in Listing 6 with inline comments.

We implemented and ran the homography transformations algorithm on both the CPU and GPU. The CPU version was implemented in C++. For evaluation, we used 60 views of the same object and generated 100 slices. A speed comparison gives a better understanding of the relative performance of CPU and GPU processors. Execution times are listed in Table 2. From this table we see that the GPU implementation operates approximately 600 times faster than the CPU implementation. In Sect. 2, we have mentioned that the hardware advantage in speed of our GPU over our CPU is approximately six times, so the increase in speed we obtained in practice requires some explanation.

We should note here that neither nVidia nor AMD disclose details about the inner structure of their processors. So, there is some inherent bias in C++ CPU to Cg GPU time comparisons. However, we can make some hypothesises about the reason why this particular algorithm worked much faster than expected.

In today's computers, the onboard installed memory (DRAM) tends to be hundreds of times slower than memory installed on both CPU caches. Therefore, program execution time heavily depends on the cache hit rate, which is typically about 90–98% for the modern computers. If we make a rough estimate of a marginal case when GPU cache hit rate for both reads and writes is equal to 100%, we find that the GPU with 100% cache hit rate will work up to a hundred times faster than the CPU with 90% cache hit rate.

There are two reasons to believe that GPU's cache hit rate is higher than the corresponding CPU's. We already stated that when the fragment processor processes texture, it cannot write into a position other than the position of a current pixel. Therefore for current pixel (x, y), the shader program can write an output only to a position (x, y) in the output texture. This is a limitation of current GPUs. However, there is a flip side to this limitation—it is excellent for the cache write optimization, because the memory write address is always known. Therefore, it is possible to attain 100% cache write hit rate for GPU. Second, because the same program is run for every pixel in the texture, it often results in a predictable pattern of memory reads. Unless you are using the conditional statements in your GPU program, memory reads have a predictable pattern. Therefore, it is natural to expect that cache read hit rate will be higher for the GPUs than for CPUs. The paper by Govindaraju et al. [12] gives some further insight into cache-effective memory models for scientific processing on GPU.

We would like also to mention here an observed speed improvement of 7,750 times over similar Matlab code. This means that while Matlab takes about 3 h to compute a single transformation, MinGPU does the same in less than 2 s! The time to load input images from a hard drive was not included; it is usually an additional 1–5 s, depending on the hard drive model, operating system state and other parameters.

If we increase or decrease the number of views or slices, the execution time increases or decreases likewise, this means that the execution time is dependent on the number of views and slices. However, tests have shown that the execution time increases exponentially if the image size exceeds a certain threshold. This threshold seems to vary depending on the video card used; therefore it is a hardware threshold. For our nVidia 7300LE graphics card, we found it at approximately 6 MB (Fig. 2). This threshold roughly corresponds to the size of the installed GPU memory cache. If the total amount of data accessed by Cg program exceeds this threshold, processing time grows according to exponential law (Fig. 2).

We can infer from this section that the algorithms that utilize homography transformations are particularly good for implementation on the GPU.

We have also acquired the latest nVidia GeForce 8800 Ultra video card and performed our experiments with the homography transformation algorithm on the same input data set. GeForce 8800 video card was installed into the same desktop computer. We have found that homographies GPU code completes in about 0.35 s on the 8800 card, which is less than a speed increase we predicted in Sect. 2. We have to note that there are many factors which contribute to the speed, such as the time required for executing C++ code on CPU, time to initialize graphics libraries,

Table 2 Comparative execution times for homography transformations

	Time per slice	Total time
MATLAB (for loops)	10.5 min	Hours
MATLAB (built-in functions)	2 min 35 s	Hours
C++ CPU	12 s	About 25 min
GPU	0.02 s	3.5 s (including 1.5 s file read)

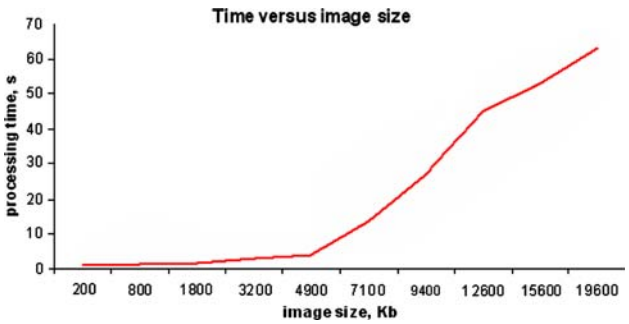

Fig. 2 Time versus image size for visual hull intersection using homography transformations

time slice taken by operating system and others. We have discovered that C++ code compiled in debug rather than release mode takes three to five times more time to execute. So, such speed comparisons are subjective.

We would like to mention here the interesting feature coming with those new cards—the CUDA technology [20]. CUDA works only with the latest 8800 series nVidia cards [19] and Tesla, which makes it the blend of software and hardware technology. CUDA allows to define shader programs ('kernels') in terms of the C code rather than in terms of shader languages like Cg, GLSL, or Sh. The CUDA toolkit contains a pre-compiler which compiles such C kernels directly into the device binary code. This code can then be linked to a C++ compilation by a host compiler like Visual Studio or sent to an 8800 video driver. Thus, users who have an access to 8800 cards are not constrained to a use of shader languages. However, these cards are now the most expensive graphics cards on market whose costs routinely exceed the costs of a desktop computer. This makes CUDA unavailable for many applications.

The other interesting features of 8800 series graphics cards are that some hardware limitations are lifted. The cards now support data scatter operations in addition to gather operations. Also, the neighboring threads are assembled into warps (currently collections of 32 threads), which can use the shared memory and synchronization services. While neither CUDA nor 8800 cards are typically capable of performing the entire computation on the GPU, and this technology does not prevent the user from knowing the intrinsic details of the GPUs, those new developments are promising and allow us to look optimistically into the future.

5 Conclusions

We have created a small library for converting existing CPU code in GPU, MinGPU, which proved to be an exceedingly useful tool in our research. Using MinGPU, we have implemented and tested a number of computer vision algorithms– homography transformations, Lukas–Kanade optical flow, correlation filters, pyramids, convolutions, and Gaussian filters on a graphics processor (GPU). Being a minimal GPU C++ library, MinGPU gives researches an opportunity to quickly test their algorithms on GPU. We have found, that using MinGPU the vision algorithms can be considerably sped up, the homography transformation algorithms in particular. Not only the GPUs offer a considerable increase in speed, they also make it possible for many algorithms to move into the real-time domain. In our work, we have also highlighted some inherent problems of contemporary GPU processors. We are now working on extension of our library to include other algorithms developed in our laboratory.

Acknowledgments We wish to express our gratitude for all the help we received in writing of this paper. We would like to thank Andrew Miller for his useful advice and help on GPU programming. The dataset and method of homography transformations was provided courtesy of Saad M. Khan. UAV IR dataset was provided courtesy of VACE CLEAR evaluation.

Appendix 1: MinGPU library

1. MinGPU class structure

MinGPU class structure is intentionally made as straightforward and small as possible. MinGPU contains two classes, *Array* and *Program*. Class *Array* defines an array, a texture, in GPU memory, while class *Program* defines a program in GPU memory. The library also includes the *MinGPUInit()* function, which is implicitly called when first MinGPU class is instantiated.

2. MinGPU methods. Class array

Class *Array* has three methods: *Create*, *CopyToGPU*, and *CopyFromGPU*.

*bool Array::Create(float *pData, unsigned int dwCols, unsigned int dwRows, BYTE bMode)*

This method defines a new array in GPU memory. The *Create* method is supplied with a pointer to an array as well as the number of columns and rows in this array. *pData* must either point to an allocated space of *dwCols* by *dwRows* size, or be *null* if the array we define will never be copied to the GPU memory. If this array is copied from the GPU memory and *pData* is *null*, then an array is created. The last parameter, *bMode*, specifies whether the array will be created in luminance or color mode. Note that array data from *pData* is not transferred to GPU in this method; it is done later, if needed, by the use of *CopyToGPU* method.

bool Array::CopyToGPU()

This method copies array data from computer memory to GPU.

bool Array::CopyFromGPU()

This method copies array data from GPU to computer memory.

3. MinGPU methods. Class program

Class *Program* has three methods: *Create*, *SetParameter*, and *Run*.

*bool Program::Create(char *szFilename, char *szEntryPoint)*

This method creates and, if needed, compiles a new Cg program. The program is stored in the external file *szFilename*. *szEntryPoint* holds the name of the entry function in the program. Programs files must be located either in working directory or in the 'scripts' subfolder. Files with 'asm' extension are presumed to contain precompiled binary programs; otherwise the file contains a source code which needs to be compiled. This function searches the folder for a binary program corresponding to the source code. If it is found and it has a timestamp later than the source file, it is loaded instead. Otherwise, the source code is compiled and stored as binary code in a file with 'asm' extension in the same folder. This eliminates the need to recompile a program each time it is loaded, which eliminates a 300–600 ms delay.

*bool Program::SetParameter(int nType, char *szName, void *pValue)*

Most functions in Cg programs have input parameters. For instance, we have to specify the input parameters for the entry function before we execute a program. Parameters can be values, arrays, matrices, or textures. Enumerator *nType* specifies the type and number format of the input parameter, string *szName* contains its Cg program name and *pValue* holds the parameter value.

Besides the parameters we set with the *SetParameter* method, functions may have parameters which refer to predefined names [6]. For example, a parameter which refers to name *TEXCOORD0* will automatically receive the row and column values for the currently processed pixel.

*bool Program::Run(Array *output)*

This method executes a Cg program on the GPU. Array *output* accepts the output of this program; it is filled as a result of program execution. The program is run separately for every cell in the array *output*. A new value is generated for every cell in this array.

All methods in classes *Array* and *Program* return true if successful and false otherwise.

4. Library structure, installation, and required libraries

The MinGPU library consists of four Visual Studio projects. The first project, *MinGPU*, contains the MinGPU library. Computer vision algorithms done on MinGPU reside in the *Vision* project. There is also a helper project, *Math*, which contains math functions used in the *Vision* project. These three projects generate C++ libraries on compilation. The fourth project, *GPUTest*, serves as a test bed for these three libraries. It generates executable code on compilation. *GPUTest* contains examples for all computer vision algorithms in the *Vision* project.

Before using MinGPU, three other libraries must be downloaded from internet and installed. These are Cg Toolkit [7], OpenGL Utility Toolkit (GLUT) and OpenGL Extensions (GLEW). A description of where to download those libraries and how they are useful in GPU computing can be found at [10]. OpenGL 2.0 drivers are supplied with Windows XP and it is important to install these also.

Appendix 2: Source listings

Listing 1 ('Hello, World'):

```
Array Array;
Program Program;
Array.Create(fpArray, cols, rows, Luminance);
Array.CopyToGPU();
Program.Create(strProgramFile, "main");
Program.SetParameter(enTexture, "texture", (void *) Array.Id());
Program.Run(&Array);
Array.CopyFromGPU();

float4 main (
   float2 coords : TEXCOORD0,
   samplerRECT texture) : COLOR
{
   float4 result;
   float4 val = texRECT(texture, coords);
   result = val + 1;
   return result; {or, equally, return 1 + texRECT(texture, coords);}
}
```

Listing 2 ('Hello, World' reduced):

```
…
Array.CopyToGPU();
Program.SetParameter(enTexture, "texture", (void *) Array.Id());
Program.Run(&Array);
Array.CopyFromGPU();
```

Listing 3 (image derivative in x, y direction):

```
…
Output.Create(NULL, cols, rows, Luminance);
Array.CopyToGPU();
Program.SetParameter(enTexture, "T", (void *) Array.Id());
 Program.SetParameter(enMatrixf, "K", (void *) Kernel.Id());
 Program.Run(&Output);
 Output.CopyFromGPU();

 float4 Derivative3x3 (
    float2 C : TEXCOORD0,
    samplerRECT T, uniform float3x3 K) : COLOR
 {
    float4 result = 0;
    for (int row = 0; row <= 2; row ++)
    {
     for (int col = 0; col <= 2; col ++)
     {
      result = result + K[row][col] * texRECT(T, C + float2(col - 1, row - 1));
     }
    }
    return result;
 }
```

Listing 4 (image derivative in t direction):

```
float4 DerivativeT3x3 (
   float2 C : TEXCOORD0,
   samplerRECT T1, samplerRECT T2, uniform float3x3 K) : COLOR
{
   float4 result = 0;
   for (int row = 0; row <= 2; row ++)
   {
    for (int col = 0; col <= 2; col ++)
    {
     float4 p1 = texRECT(T1, C + float2(col - 1, row - 1));
     float4 p2 = texRECT(T2, C + float2(col - 1, row - 1));
     result = result + K[row][col] * p2 - K[row][col] * p1;
    }
   }
   return result;
}

float4 DerivativeT1x1 (
   float2 C : TEXCOORD0,
   samplerRECT T1, samplerRECT T2) : COLOR
{
   return  texRECT(T2, C + float2(col, row)) - texRECT(T1, C + float2(col, row));
}
```

Listing 5 (pyramid REDUCE operation):

```
…
Array.CopyToGPU();
Mask.CopyToGPU();
Program.SetParameter(enTexture, "T", (void *) Array.Id());
Program.SetParameter(enArrayf, "Mask", (void *) Mask.Id());
Program.Run(&Output);
Output.CopyFromGPU();

float4 ReducePyramid  (
   float2 C : TEXCOORD0,
   samplerRECT T, uniform float Mask[5][5]) : COLOR
{
   float4 result = 0;
   for (int row = 0; row <= 4; row ++)
   {
    for (int col = 0; col <= 4; col ++)
    {
     result = result + Mask[row][col] * texRECT(T, float2(2 * C.x + col - 2, 2 * C.y + row - 2));
    }
   }
   return result;
}
```

Listing 6 (homography transformation):

```
{Initialize script. Set script parameters that won't change later}
Script Script;
Script.Create(strScript_Homography, "main");
Script.Select();
Script.SetParameter(enFloat, "ratio", (void *) &fRatio);
Script.SetParameter(enFloat, "crop_x1", (void *) &crop_x1);
Script.SetParameter(enFloat, "crop_y1", (void *) &crop_y1);

{For every slice}
for (int slice = 1; slice <= SLICES; slice++)
{
 {Create and load empty output texture}
 Texture Accumulator;
 Accumulator.Create(fb  uf, crop_x, crop_y);
 Accumulator.CopyToGPU();
 {Bind script to this texture}
 Script.SetParameter(enTexture, "result", (void *) & Accumulator)
  {For every view}
  for (int i = 1; i <= VIEWS; i++)
  {
        {set script parameters - view and homography matrix}
        Script.SetParameter(enTexture   , "view", (void *) &View);
        Script.SetParameter(enMat   rixf, "H", &HMatrix);
        {run the script}
        Script.Run(&Accumulator);
  }
  {recover results from the script}
  Accumulator.CopyFromGPU();
}

{Cg Script}
float4 main (
 {current coordinates in cropped region}
 float2 coords : TEXCOORD0,
 {3x3 homography transformation matrix}
 uniform float3x3 H,
 {(x, y) gives offset of cropped region in target image}
 uniform float ratio, uniform float x, uniform float y,
 {view refers to input view image, result refers to accumulator image}
 uniform samplerRECT view, uniform samplerRECT result) : COLOR
{
 {Input coordinates, coord, contain coordinates in cropped region, 200x180}
 {To do proper transformation, we have to convert coords to target image}
 float3 C = float3(coords[0] * ratio + x, coords[1] * ratio + y, 1.0);
 {the next three lines do homography transformation}
 float k = H[2][0] * C[0] + H[2][1] * C[1] + H[2][2] * C[2];
 float A = (H[0][0] * C[0] + H[0][1] * C[1] + H[0][2] * C[2]) / k;
 float B = (H[1][0] * C[0] + H[1][1] * C[1] + H[1][2] * C[2]) / k;
 {now take already accumulated value from the result …}
 float4 x = texRECT(result, coords);
 {… and add a new value to it}
 float4 y = texRECT(view, float2(A, B));
 return x + y;
}
```

References

1. Adelson, E.H., Anderson, C.H., Bergen, J.R., Burt, P.J., Ogden, J.M.: Pyramid methods in image processing. RCA Eng. 29(6), 33–41 (1984)
2. Buck, I., Foley, T., Horn, D., et al.: Brook for GPUs: stream computing on graphics hardware. ACM. Trans. Graph. **23**(3), 777–786 (2004)
3. BrookGPU source code: http://brook.sourceforge.net
4. Colantoni, P., Boukala, N., Da Rugna, J.: Fast and accurate color image processing using 3D Graphics Cards. In: Vision, Modeling and Visualization Conference (VMV) (2003)
5. Computer Vision on the GPU: http://openvidia.sourceforge.net/papers.shtml.
6. Cg Toolkit User's Manual. (2005)
7. Cg Toolkit: http://developer.nvidia.com/object/cg_toolkit.html
8. Fung, J.: GPU Gems 2, chapter Computer Vision on the GPU. Addison Wesley, pp. 649–666 (2005)
9. Fung, J., Mann, S., Aimone, C.: OpenVidia: parallel GPU computer vision. Proceedings of ACM Multimedia, pp. 849–852 (2005)
10. Göddeke, D.: Online tutorial on GPGPU. http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html
11. Govindaraju, N.K., Manocha, D.: Cache-efficient numerical algorithms using graphics hardware. UNC Technical Report (2007)
12. Govindaraju, N.K., Larsen, S., Gray, J., Manocha, D.: A memory model for scientific algorithms on graphics processors. UNC Technical Report (2006)
13. Han, Y., Wagner, R.A.: An efficient and fast parallel-connected component algorithm. J. ACM. 37 (3), 626–642 (1990)
14. Heymann, S., Müller, K., Smolic, A., Fröhlich, B., Wiegand, T.: SIFT implementation and optimization for general-purpose GPU. In: International Conference on Computer Graphics, Visualization and Computer Vision (WSCG) (2007)
15. Khan, S.M., Yan, P., Shah, M.: A homographic framework for the fusion of multi-view Silhouettes. In: International Conference on Computer Vision, Rio de Janeiro, Brazil (2007)
16. Labatut, P., Keriven, R., Pons, J-P.: Fast level set multi-view stereo on graphics hardware. In: 3rd International Symposium on 3D Data Processing, Visualization and Transmission (3DPVT), pp. 774–781 (2006)
17. MinGPU source and technical report (case-sensitive): www.cs.ucf.edu\~vision\MinGPU
18. Moreland, K., Angel, E.: The FFT on GPU. SIGGRAPH, pp. 112–119 (2003)
19. NVIDIA GeForce 8800 GPU Architecture Overview. nVidia Technical brief (2006). www.nvidia.com/object/IO_37100.html
20. NVIDIA CUDA Programming Guide 1.1. http://www.nvidia.com/object/cuda_home.html.
21. OpenVIDIA GPU computer vision library. http://openvidia.sourceforge.net
22. Ohmer, J., Maire, F., Brown, R.: Implementation of Kernel methods on the GPU. In: Proceedings 8th International Conference on Digital Image Computing: Techniques and Applications (DICTA). pp. 543–550 (2005)
23. RapidMind development platform and documentation, http://www.rapidmind.net/.
24. Rumpf, M., Strzodka, R.: Level set segmentation in graphics hardware. In: Proc. IEEE Int. Conf. Image Process. **3**, 1103–1106 (2001)
25. Sh GPU metaprogramming library: http://www.libsh.org/
26. Yang, R., Pollefeys, M.: Multi-resolution real-time stereo on commodity graphics hardware. IEEE Comp. Vis. Pattern Recog. 211–218 (2003)
27. Yang, R., Welch, G.: Fast image segmentation and smoothing using commodity graphics hardware. J. Graph. Tools **7**(4), 91–100 (2002)

Author Biographies

Pavel Babenko received his BS degree in Computer Science from Belarussian State University of Informatics and Radioelectronics, Minsk, Belarus, in 1998 and his MS degree in Computer Science from University of Central Florida in 2006. From 1998 to 2004 he has worked in information technology industry. Currently, he is working toward the Ph.D. degree at the Computer Vision Laboratory at the University of Central Florida. His current research interests include general-purpose computations on GPU, multiple view geometry, statistical pattern recognition and application of computer vision methods to bioinformatics. He is a student member of the IEEE.

Mubarak Shah received the B.E. degree in electronics from Dawood College of Engineering and Technology, Karachi, Pakistan, in 1979. He completed the E.D.E. diploma at Philips International Institute of Technology, Eindhoven, The Netherlands, in 1980, and received the M.S. and Ph.D. degrees in computer engineering from Wayne State University, Detroit, MI, in 1982 and 1986, respectively. He is the Chair Professor of Computer Science and the founding Director of the Computer Visions Laboratory, University of Central Florida (UCF), Orlando. He is a Researcher in a number of computer vision areas. He is a Guest Editor of the special issue of International Journal of Computer Vision on Video Computing. He is an Editor of the international book series on Video Computing, the Editor-in-Chief of the Machine Vision and Applications Journal, and an Associate Editor of the Association for Computer Machinery (ACM) Computing Surveys Journal. He was an IEEE Distinguished Visitor Speaker from 1997 to 2000. Dr. Shah is the recipient of the IEEE Outstanding Engineering Educator Award in 1997. He also received the Harris Corporation's Engineering Achievement Award in 1999, the Transfer of Knowledge Through Expatriate Nationals (TOKTEN) Awards from the United Nations Development Programme (UNDP) in 1995, 1997, and 2000, the Teaching Incentive Program Awards in 1995 and 2003, the Research Incentive Award in 2003, Millionaires' Club Awards in 2005 and 2006, the University Distinguished Researcher Award in 2007, Honorable Mention for the International Conference on Computer Vision (ICCV) 2005, and was also nominated for the Best Paper Award in the ACM Multimedia Conference in 2005. In 2006, he was awarded the Pegasus Professor Award, the highest award at the UCF. He was an Associate Editor of the IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI).