GpuCV: A GPU-accelerated framework for Image Processing and Computer Vision

Yannick Allusse, Patrick Horain, Ankit Agarwal, and Cindula Saipriyadarshan

Institut TELECOM; TELECOM & Management SudParis
9, rue Charles Fourier, 91011 Evry Cedex, France
Patrick.Horain@IT-SudpParis.eu

Abstract. This paper presents briefly describes the state of the art of accelerating image processing with graphics hardware (GPU) and discusses some of its caveats. Then it describes GpuCV, an open source multi-platform library for GPU-accelerated image processing and Computer Vision operators and applications. It is meant for computer vision scientist not familiar with GPU technologies. GpuCV is designed to be compatible with the popular OpenCV library by offering GPU-accelerated operators that can be integrated into native OpenCV applications. The GpuCV framework transparently manages hardware capabilities, data synchronization, activation of low level GLSL and CUDA programs, on-the-fly benchmarking and switching to the most efficient implementation and finally offers a set of image processing operators with GPU acceleration available.

Keywords: GPGPU, GLSL, CUDA, computer vision, image processing

1 Introduction

Graphical processing units (GPUs) are powerful parallel processors mostly dedicated to image synthesis. They have made their way to consumers PCs through video games and multimedia. Recent graphics cards generation[] offer highly parallel architectures (hundreds of processing units) with high memory bandwidth and are close to reach TeraFLOPS peak performances while CPUs barely reach 50 GigaFLOPS, but they suffer from complex integration and data manipulation procedures based on dedicated APIs. While they have become the most powerful part of middle-end computers, they opened a path to cheap General Purpose processing on GPU (GPGPU).

In this paper, we discuss the benefits and issues related with GPU for image processing. Then we describe GpuCV, the open source framework we propose for GPU-accelerated image processing and computer vision, which is an extension of OpenCV[1], a popular library for developing interactive computer vision applications.

The GpuCV framework is meant to transparently manage hardware capabilities across card generations, data synchronization between central and graphics memory and activation of low level GLSL and CUDA programs.

It performs on-the-fly benchmarking and switching to the most efficient implementation depending on operator parameters.

Finally, we introduce the set of image processing operators with GPU acceleration available in GpuCV and we discuss how to move existing OpenCV applications to GpuCV.

2 GPU Processing

2.1 GPU technologies

Graphics processors are meant to generate high quality images from three-dimensional models. Their massively parallel processing pipeline offers great abilities for algebra operations and native handling of floating point numbers, vectors and matrices. Their host graphics card holds dedicated fast memory. General purpose computing with GPU is possible at the cost of reformulating common algorithms to fit the processing pipeline, which may or may not be easy depending on the algorithm.

Since 2002, GPUs have become programmable pipelines with vertex and fragment shaders written in Cg (C for Graphics)[2], GLSL (OpenGL Shading Language)[3] and HLSL (High Level Shading Language). While these languages require compiling and linking GPU and CPU programs, the meta-programming approach in Sh[4] and Brook GPU[5] is meant to hide difference between CPU and GPU.

In the end of 2006, NVIDIA released a new architecture named CUDA (Compute Unified Device Architecture)[6] and AMD/ATI released CTM (Close To Metal)[7], which both enhance GPU control.

All this programming languages have helped turning GPUs into data stream processors, so leaving the central processing unit (CPU) available for other tasks. The challenge consists of creating shaders to achieve the expected processing, while working around their intrinsic limitations. Although, they are not meant for image processing and remain hard to use for image processing and computer vision scientists.

2.2 Previous work on GPU computing for image processing

The 1st publications we are aware of that describe image processing with consumer GPU are dated 1999 [8] and open source libraries became available in 2003[9]. Colantoni[10] used GPU programming for color image processing. Jargstorff [11] proposed a framework for image processing on GPU that was implemented byNocent[12]. The OpenVidia library [13] by *Fung et al.* offers a library of fragment shaders and a framework for computer vision and image processing. In [14], Moreland and Angel used a fragment shader to compute a Fast Fourier Transform on GPU four times faster than on CPU. Strzodka presented a GPU-accelerated generalized bi-dimensional distance transform in [15] and motion estimation[16]. The GPUGems books serie[17] discusses image processing, including image filtering(color adjustment, anti aliasing), image processing in the

OpenVidia framework, and advanced GPGPU programming Gaussian computation. Lately, GPU4Vision[18] achieved real-time optical flow and total variation based image segmentation on GPU.

3 GPU caveats

General purpose computing with GPUs implies some challenges and technological issues.

3.1 Platform dependency

GPU technologies are evolving rapidly and rely on dedicated interfaces meant for parallel image rendering. Every year, a new generation of graphics chipset is released with new features, extensions and backward compatibility issues such as shader model version, image formats available (double float precision), NVIDIA CUDA or ATI CTM support.

3.2 Data transfers

Mixed processing on CPU and GPU implies data transfers between the central memory (CPU RAM) and the video memory (GPU RAM), which can turn into a bottleneck. Running several operators consecutively on either processor helps reducing the transfer cost, so enhancing acceleration. At times, an operator may be slower on the processor that holds data in memory, but processing locally avoids data transfer and so may achieve better overall performance.

3.3 Sequential to parallel processing

While parallel algorithms for image processing, that process each pixel independently, can be fairly easily implemented on GPU, sequential algorithms are well suited for the CPU architecture. Some sequential algorithms can hardly be transposed efficiently on the GPU parallel architecture. Global image computation (histogram, labeling, distance transform, recursive filters, sum array table) requires *ad hoc* implementation. Recent technology such as CUDA helps but still requires tricky tuning for efficient acceleration[19].

3.4 Varying relative GPU/CPU performances

Activating a code fragment on GPU requires a code dependent activation delay, so processing small images is not accelerated with GPUs. Indeed calling a program on the GPU has an overhead cost (about 100 micro-sec for CUDA, 180 micro-sec for OpenGL and GLSL) that can be more than the CPU operator execution time. Furthermore, hiding the latency of the GPU memory requires processing some minimum amount of data to increase the number of consecutive threads that are executed in parallel. Performance of operators may also vary depending on data size and format. A GPU routine may run faster with large images than its CPU counterpart, but slower with small images.

3.5 API restrictions

The output of a fragment shader is write-only, i.e. it cannot be read by that shader, so recursive algorithm mist implemented with successive calls of a shader. NVIDIA CUDA work around theses limitations at the cost of a more complex data format management. Indeed, CUDA achieves direct access to the graphics card so pixel format conversions previously done by the graphic drivers now must be handled at the application level[19].

4 GpuCV approach

We propose GpuCV as an open source library and framework for image processing and computer vision accelerated by GPU. It is meant to support developers that may be familiar or not with the GPU technology in achieving GPU acceleration by:

- offering a set of GPU optimized routines in replacement of some OpenCV library routines.
- transparently comparing the CPU and GPU implementations and automatically switching to the fastest one.
- hiding platform dependencies and data transfers.

We describe here after the main GpuCV framework features, namely the application programming interfaces(API) it relies on, the data management and implementation switching mechanisms and finally the available facilities for integration into existing applications.

4.1 Programming technologies

GpuCV supports an OpenGL-GLSL and a NVIDIA CUDA API, in order to get the best from each. OpenGL and GLSL have been widely used and feature high compatibility with most hardware (AMD/ATI, NVIDIA, Intel, S3) and OS. The OpenGL-GLSL API uses general OpenGL rendering features such as rendering-to-textures, depth buffer, mipmapping as well as vertex/geometry/fragment shaders for custom operations. It allows 2D/3D contents computing and makes abstraction of the data types and formats. The GpuCV-CUDA API relies on the CUDA[] library which allows recursive operators but is currently compatible only with recent NVIDIA graphics cards. Most operators supplied with GpuCV have been developed with both API for compatibility purpose.

GpuCV does not use meta-programming languages such as BrookGPU to stay closer to the hardware and have full control of memory management and graphics cards features.

4.2 Data management

Processing data with either CPU or GPU requires storing data in central memory and/or in graphics memory. Furthermore data may have to be available in several formats in either memory, e.g. as IplImage or CvMat for OpenCV in central memory, or as texture of buffer for OepnGL and array or buffer for CUDA in graphics memory. Handling data potentially stored in multiple locations and formats requires synchronizing copies and enforcing read only access to input images. GpuCV saves developers the burden of managing data with a unified data container that describes the data format of an image and allows transparent data handling. In case the data location or format does not match the selected implementation, the image is transparently copied into the required location and formats.

In case data is available from several locations, a 'smart transfer' option can estimate all possible transfer time costs and select the fastest one, based on previously recorded benchmarks. Finally, GpuCV operators know about input and output images, so writing to an output image discards all the other existing instances for data consistency sake.

4.3 Automatic switching a GpuCV operator

A GpuCV enhanced application should run on a CUDA enabled platform, or an older GLSL only platform or even a low end CPU only platform. So a GpuCV operator may include up to three implementations respectively based on OpenCV, OpenGL-GLSL and NVIDIA-CUDA. Obviously, the execution time of each implementation depends on the algorithm, on the input parameters such as image size and format, on the optional filter parameters and the host hardware platform. So performance depends on too many parameters to be easily predicted and no implementation can be statically chosen as the fastest for any operator. Second, they require data in associated memory (central or graphic memory) and data transfer might be done according to the previously used implementation and add more complexity to already complex source code. We have developed a dynamic switch mechanism that works heuristically based on local implementations' benchmarks and estimated transfer times. We have implemented this mechanism internally to each GpuCV operator to transparently switch between the CPU and GPU implementations.

Switch implementation: The switch mechanism performs in the following three modes:

- Benchmarking mode - Collects, on the fly, processing times for all implementations.
- Switch mode - Chooses best implementation to call depending on previously recorded benchmarks.
- Forced mode - User can force the switch to call any of the implementations.

Compatibility of the workstation hardware with an implementation is respected by the switch in all modes. Also to ensure full compatibility with the native CPU operator we synchronize input data to CPU memory when required.

Benchmarking mode runs until we get significant information about all implementations according to their input parameters such as image properties and optional operator parameters. We use *SugoiTracer*[20] to collect the statistics (such as average processing time, standard deviation, total time...). The mechanism leaves benchmarking mode to go to switch mode when the standard deviation time shows stable and coherent values.

In the switch mode, it calculates the calling cost for each implementation using the processing time and eventual data transfer time depending on the data memory location. Then it calls the fastest implementation.

Finally the switch can be forced by the user to call a desired implementation for any operator. It can be used to select an implementation for show case or benchmarks as well as to avoid the switching cost for small images.

Converting all OpenCV operators to GpuCV auto-switch operators: GpuCV supplies several interfaces to directly access all the GPU implementations from GpuCV-GLSL and GpuCV-CUDA as well as a switching interface which contains all the switch operators. The switching interface is self generated using OpenCV functions' declarations and uses dynamic library loading mechanism to find all GpuCV available implementations. Knowing the auto-switch has an observed mechanism time of about 350μs, which is negligible for large images but become too costly for really smaller ones. As all the GpuCV interfaces respect OpenCV original functions declarations, developers can either directly call implementations at the cost of some manual optimization and synchronization or simply call the auto-switch operators to ensure that the fastest implementations is called.

4.4 Integration

GpuCV has been designed to be fully compliant with existing OpenCV applications, and thus on multiple OS such as MS Windows XP and LINUX.

Porting an OpenCV application to GpuCV: As previously described, the smart data transfer mechanism transparently handles multiple data locations and formats and the automatic switch mechanism select the most efficient implementation available. This makes it possible to smoothly and easily integrate GPU acceleration routines for the GpuCV library with CPU based routines from the OpenCV popular library[1]. Actually, the highest level interface to GpuCV is a set of routines that are meant as replacement for OpenCV native routines. Porting an existing OpenCV application to GPU now consists of changing a few header files, linking libraries and adding manual synchronization when image data are accessed without using OpenCV functions.

More advanced manual tuning of GpuCV can be achieved whose explanation are described in on-line documentation.

5 Results

This section present our results, all source code and benchmarks are available on-line. Testing hardware was an Intel Core2 Duo 2.13 Ghz CPU with 2GB of RAM and NVIDIA GeForce GTX280 GPU with 1GB of RAM.

5.1 Benchmarking tools

GpuCV integrates some embedded benchmarking tools[20] that are used to record data transfer times and processing time for all GPU and CPU implementations. It can be used to benchmark a native OpenCV application and return statistics about all the OpenCV calls depending on input parameters such as data size, format and operators options such as filter size of filter mode.

5.2 Point to point operations

GpuCV includes numerous point to point operations for arithmetic, logic, comparison and math functions. Their implementations are straight forwards and will not be discussed here.

5.3 Neighborhoods operations

GpuCV supplies neighborhoods operators for morphology and edges detection such as dilate, erode, sobel, laplace, canny and smooth. They have similar implementations except GpuCV-CUDA versions that use shared memory as cache for fast access[6]. Table 1 shows some results.

Table 1. Benchmarks for some neighborhood operators.

Operator	Erode/Dilate 3			Sobel		
Image Size	OpenCV	GpuCV-GLSL	GpuCV-CUDA	OpenCV	GpuCV-GLSL	GpuCV-CUDA
2048×2048	30ms	2.68ms	1ms	49.2ms	13.7ms	1.1ms
1024×1024	7.45ms	$860\mu s$	$390\mu s$	13.5ms	3.6ms	$400\mu s$
512×512	4.7ms	$342\mu s$	$160\mu s$	3.7ms	1.1ms	$176\mu s$
256×256	1.9ms	$224\mu s$	$115\mu s$	$915\mu s$	$413\mu s$	$117\mu s$
128×128	$122\mu s$	$211\mu s$	$110\mu s$	$240\mu s$	$245\mu s$	$113\mu s$

5.4 Sequential algorithm

The Deriche[21] filter is an edge detection filter. It performs a 2 ways smoothing on X and derivation on Y, 2 ways smoothing on Y and derivation on X and finish by a local extrema calculation. Deriche is more efficient than canny or sobel filters but requires more calculation so it is often used in real time applications. The processing of one line/row is sequential but all the lines/rows can

be processed in parallel so we have decided to port Deriche to GpuCV-CUDA. The default algorithm for smoothing and derivation is to scan line Xn from left to right and store results into XnL, and from right to left into XnR, then it sums both temporary lines XnL and XnR into Xn_Sum. XnL and XnR calculations are independent and can be done in parallel using different thread IDs. To avoid wasting memory with temporary buffers (XnL and XnR) and waiting the calculation end to sum results, we performs a two steps pass for X:

- For left to right processing:
  - $i \in [0, width/2] : Xn\_Sum[i] = XnL[i]$
  - $i = width/2$: we synchronize threads
  - $i \in [width/2, width] : Xn\_Sum[i] = Xn\_Sum[i] + XnL[i]$
- For right to left processing:
  - $i \in [width, width/2] : Xn\_Sum[i] = XnR[i]$
  - $i = width/2$: we synchronize threads
  - $i \in [width/2, 0] : Xn\_Sum[i] = Xn\_Sum[i] + XnR[i]$

To optimize memory access, only row are processed so it requires the transposed input images. As smoothing Y is independent from from smoothing X, a single kernel process simultaneously *Img* and *Transposed(Img)* depending on the thread IDs. Derivations are identically handled and local extrema calculation is done by a simple kernel. As we have only a few threads running (2 per lines + 2 per rows), the GPU is not in full charge for small images. Table 5.4 shows real improvements up to 100 times so GPU based Deriche filter is now able to perform on large images in real time.

Table 2. Benchmarks for Deriche

Image Size	Cimg	GpuCV-CUDA	Improvement factor
2048×2048	1997ms	19.35ms	101
1024×1024	397ms	6.58ms	60
512×512	89.2ms	3.625ms	24
256×256	26.5ms	2.907ms	9
128×128	7.07ms	2.8ms	2.5

5.5 Object detection

As a further example, we try to implement on GPU the object detector algorithm described by *Viola*[22]. A trained classifier (namely a cascade of boosted classifiers working with haar-like features) is applied to the image's region of interest, and return a position when the region consist of a Haar-like feature. *Viola*'s object detection algorithm involves two operations, Summed Area Table(SAT) and Local Area Sum. We implemented both on GPU, SAT based on CUDPP library[23] and the local sum by a simple kernel. Refer to Table 5.5 for

results. So, our next step would be to implement ***CvHaarClassifierCascade*** tree data structure on GPU memory by organizing the dispersed data of a tree into a continuous data on GPU and remapping the data pointers. Half of the processing time of OpenCV operator is spent in local sum calculation. Also the transfer time which is wasted in transferring data from CPU to GPU is covered by the gain from SAT. With this implementation, we expect to get better results than OpenCV.

Table 3. Benchmarks for SAT and LocalSum

Operator	SAT		LocalSum	
Image Size	OpenCV	GPUCV	OpenCV	GPUCV
2048×2048	78.7ms	27.7ms	31.8ms	1.4ms
1024×1024	22.5ms	5.4ms	7.8ms	$893\mu s$
512×512	4.9ms	2.7ms	1.9ms	$198\mu s$
128×128	142 μs	2.4ms	$133\mu s$	$105\mu s$

6 Conclusion

In this paper, we presented benefits and issues of using GPGPU for image processing. We described our open source framework for image processing and computer vision, which is an extension of the Open CV library. It is meant to help scientist and developer porting their existing applications or new algorithm GPU without falling into low level GPU complexity. It offers many features to transparently manage hardware capabilities, data synchronization, GLSL and CUDA support, on-the-fly benchmarking and switching mechanisms and finally offers a set of image processing operators with GPU acceleration available.

We shown that GPU-acceleration depends from many factors namely the algorithm to optimize, the image properties and format, the hardware and the optional filter parameters, so it is hard to predict the performance on all systems. Furthermore, some cases remain faster on CPU than GPU so we introduce a dynamic mechanism to select the most efficient implementation and reduce GPU usage penalties. Future works will consist on adding more operators to the library and improving integration with other image processing systems.

As an open source project, we encourage the community to use and contribute to the library. GpuCV sources, benchmarks and informations are available at: https://picoforge.int-evry.fr/projects/gpucv/.

References

1. SourceForge.net: Open Computer Vision Library. (http://sourceforge.net/projects/opencvlibrary)

2. Fernando, R., Kilgard, M.: The Definitive Guide to Programmable Real-Time Graphics. Addison-Wesley Longman Publishing Co. (2003)
3. Rost, R.: OpenGL Shading Language. Addison-Wesley professional (2004)
4. Mac Cool, M., Du Toit, S.: Metaprogramming GPUs with Sh. AK Peters, Inc (2004)
5. Buck, I., al.: BrookGPU. http://graphics.stanford.edu/projects/brookgpu/ (2003)
6. NVIDIA: CUDA (Compute Unified Device Architecture). http://www.nvidia.com/object/cuda_home.html (2006)
7. AMD/ATI: CTM (Close To Metal), (2007). (http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf)
8. Hopf, M., Ertl, T.: Hardware-Based Wavelet Transformations. Workshop of Vision, Modelling, and Visualization (VMV '99) (1999) 317–328
9. Yaromenok, A.: DIPlib - Digital Image Processing Library. http://sourceforge.net/projects/diplib (2003)
10. Colantoni, P., Boukala, N., Da Rugna, J.: Fast and accurate color image processing using 3d graphics cards. 8th International FallWorkshop: Vision Modeling and Visualization, Munich, Germany (2003)
11. Jargstorff, F.: A framework for image processing. In: GPU Gems volume 1. Addison Wesley professional (2004) 445–467
12. Nocent, O.: Image processing with OpenGL. http://leri.univ-reims.fr/nocent/gpu.html (2004)
13. Fung, J., al.: OpenVIDIA: Parallel GPU Computer Vision. http://openvidia.sourceforge.net (2004 and later on)
14. Moreland, K., Angel, E.: The FFT on a GPU. SIGGRAPH/Eurographics Workshop on Graphics Hardware (2003) 112–119
15. Strzodka, R. Telea, A.: Generalized Distance Transforms and skeletons in graphics hardware. In: IProceedings of EG/IEEE TCVG Symposium on Visualization (VisSym04) (2004), pp. 221-230. (2004)
16. Strzodka, R., Garbe, C.: Real-time motion estimation and visualization on graphics cards. In: Proc. IEEE Visualization. (2004) 545–552
17. GPU Gems Vol 1., 2., 3.: Programming Techniques for High-Performance Graphics and General-Purpose, Addison Wesley professional (2004,05,07)
18. GPU4Vision: . http://www.gpu4vision.org (2008)
19. Harris, M.: SC07-High Performance Computing with CUDA-Optimizing CUDA. (http://www.gpgpu.org/sc2007/SC07_CUDA_5_Optimization_Harris.pdf)
20. Allusse, Y.: SugoiTracer. http://sugoitools.sourceforge.net/ (2006)
21. Deriche, R.: Fast algorithms for low-level vision. In: 9th International Conference on Pattern Recognition, 2007. NSS '07. IEEE, Rome. Volume 4. (1988) 434–438
22. Viola, P., Jones, M.: Rapid object detection using a boosted cascade of simple features. In: Proc. IEEE Computer Society Conference on Computer Vision and Pattern Recognition CVPR 2001. Volume 1. (2001) 511–518
23. NVIDIA: CUDPP(CUDA Data Parallel Primitives Library). http://www.gpgpu.org/developer/cudpp/ (2006)