High Performance Realtime Vision for Mobile Robots on the GPU

Christian Folkers and Wolfgang Ertel

University of Applied Sciences Ravensburg-Weingarten, Germany

Abstract. We present a real time vision system designed for and implemented on a graphics processing unit (GPU). After an introduction in GPU programming we describe the architecture of the system and software running on the GPU. We show the advantages of implementing a vision processor on the GPU rather than on a CPU as well as the shortcomings of this approach. Our performance measurements show that the GPU-based vision system including colour segmentation, pattern recognition and edge detection easily meets the requirements for high resolution (1024×768) colour image processing at a rate of up to 50 frames per second. A CPU-based implementation on a mobile PC would under these constraints achieve only around twelve frames per second. The source code of this system is available online [1].

1 Introduction

The Robocup competitions of the last few years have shown that efficient image processing is on of the essential requirements for a successful team. Although this requirement is well known, many teams still suffer from serious performance problems of the image processing on their mobile computers. At the first glance, this seems somewhat surprising, because the image processing algorithms required for solving the tasks in the Robocup middle-size league are well known. However, there is a simple answer. Besides the high speed of the movements on the playground, the rules of the game require that each robot is autonomous. Therefore each robot needs its own image processing on the robot, what leads to problems with weight, space, energy consumption, and computing power on the mobile PC of the robot. The requirements on mobile robots for industrial tasks are often very similar.

In order to perform image processing on a mobile CPU in real time, the algorithms often have to be strongly simplified. This leads to high noise, inexact localisation and object classification and often difficult and errorprone manual calibration.

Thus, image processing for middle-size league Robocup robots still is a crucial and interesting challenge, with yet only partial success.

1.1 Our Approach

High resolution colour image processing in real time on mobile systems not only requires vector-based fast floating point computations on, but also very high storage access bandwidth for reading and writing of image data. State of the art mobile CPUs are not fast enough to meet these requirements.

Our Approach solves this problem by means of the fragment processor, a coprocessor of the GPU which is specialised on simple number crunching tasks.

1.2 State of the Art

Powerful graphics boards with passive cooling are commercially available only recently and the exploitation of their capabilities for image processing requires advanced knowledge. Hence, comprehensive libraries for image processing on the GPU are still missing.

All available libraries are running completely on the CPU, but they are not fast enough for a mobile realtime system. There is one exception, the OpenVIDIA library [6].

OpenVIDIA is a research project aiming at the implementation of a complete library for high-level image processing in real-time on the GPU. The current version of this library however requires six GPU boards in parallel and thus in the near future it will not be available for mobile systems.

The only realistic alternative that is applied in the Robocup community is the utilisation of FPGAs (field programmable gate arrays) or even special purpose chips. These programmable logic circuits resemble older GPUs with the difference that they have no direct connection to the main board via an internal bus. Rather they have to be accessed via external interfaces. Furthermore, access to the main memory of FPGAs and chips is relatively slow as compared to modern GPUs with highly parallel memory access with optimized special caches on board. Finally, FPGAs and special purpose chips are not commercially available for standard PCs and notebooks.

2 GPU Architecture

The GPU is a special purpose microprocessor on the graphics board which is connected via an AGP or PCI-express interface with the main board. Being designed for computationally large vector based floatingpoint operations the GPU with its SIMD (single instruction multiple data) architecture significantly differs from the CPU architecture (c.f. figure 1).

After its transfer from the CPU to the GPU, a program is being executed for a large number of pixels in parallel in between 8 and 24 parallel pixel-pipelines. The so called fragment program has the limitation that each one of its parallel instances has no access on data of other instances. This parallel execution without communication results in a linear speedup between 8 and 24.

In order to avoid a memory access bottleneck, the GPU is mounted without apron directly on the board and connected via a 128 to 256 bit wide bus to the memory. Depending on the type of memory used (DDR, DDR2 or even DDR3 with RAMDACs between 400 and 1000 MHz), transfer rates between 16 and 35 GByte/sec can be achieved.

Because of the parallel execution of one program on many contiguous pixels, memory access can be accelerated considerably with special purpose caches. This caching mechanism is optimized for sequential reading and strongly differs from standard algorithms (c.f. figure 2). It is implemented in hardware and thus can not be modified by the programmer.

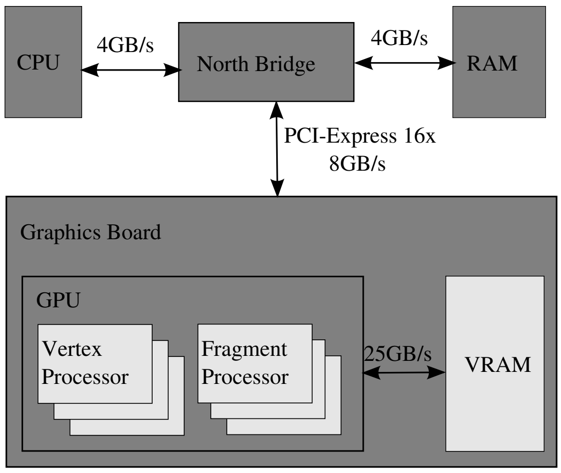

Fig. 1. The hardware architecture.

	GPU(FP)	CPU
architecture	SIMD	CISC
number of instructions	33	$\approx$ 400
processor clock rate [MHz]	350-400	2500-3500
degree of parallelism	8x-16x	$\approx$ 3x
processing speed [GFLOPS]	40-53	12
memory size [MB]	128-256	512-1024
memory clock rate [MHz]	800-1000	400-533
memory bandwidth [GB/sec]	16-35	4-6

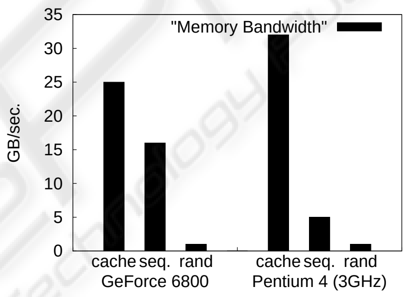

Fig. 2. Comparison of GPU and CPU performance (left) and memory bandwidth (right) [3]. Sequential reading at a rate of 16 GB/sec. is much faster on the GPU.

As another special feature, the fragment processor is designed to work on four dimensional vectors of 32 bit floating point numbers (with appropriate registers).

Due to the great differences between the architectures of CPU and GPU, it is hard to directly compare them. Therefore, in table 2 we compare the CPU with the fragment processor, which is only one part of the GPU. The values for the CPU correspond to a Pentium 4 processor and for the GPU to a NVIDIA GeForce 6800 board.

3 GPU Programming

Unlike the CPU the GPU is not a general purpose processor and thus can not be programmed with a universal programming language like C or C++. The GPU uses fixed-function pipelines which can only be configured and two free programmable processors (starting with GeForce 8 there are three of them):

- The vertex processor for the manipulation of geometry data by means of vertex-shaders.
- The fragment processor for pixel operations with pixel-shaders.

For calling functions on the GPU from the CPU, the standardized free and vendor independent graphics library OpenGL [7] by SGI is being used.

3.1 Fragment Processor Programming

The GL_ARB_fragment_program extension allows for programming the fragment processor in assembler. Obviously this way of programming is very time consuming and does not scale well with improvements of the GPU. Therefore, for most of the programs in our image processing library we used GLSL (OpenGL Shading Language), a high level language, similar to C. As a part of the extension GL_ARB_fragment_shader, GLSL has been developed by 3Dlabs. Later on the ARB (Architecture Review Board) announced it as a standard. An extensive specification is available online [8].

A special feature of this language is that compilation of programs is performed during run-time by the driver of the GPU. This makes it possible that existing programs can benefit from new hardware features or better compilers.

4 Architecture Overview

Our OrontesNG low-level vision module (short ONG vision module) was designed for low-level image processing in real-time. Tasks such as Canny Edge Detection, Gaussian Blur, RGB Colour Normalization or even simple conversion from RGB to HSV typically require little logic, but enormous computing power because they are processed completely on the pixel level. To optimize performance, these operations have been implemented on the GPU. Please note that normally the GPU is used to process the enormous amount of graphics data of modern video games. The data flow there goes from CPU to GPU only, whereas in our image processing application, the reverse direction from the GPU to the CPU is important as well. The ONG vision module is a C++ library with an API for image processing tasks like the ones mentioned above (see Figure 3).

Though the ONG vision library is implemented as singleton, it can process tasks from an arbitrary number of threads in parallel. During the initialization the maximum image size must be specified, such that internal buffers can be allocated.

After initialisation of the module jobs can be generated and via $Invoke()$ functions can be called. These jobs then are executed in a FIFO strategy by the GPU. Once started, a job can not be cancelled before termination.

There are two different methods to wait for the termination of a job. First, there is a polling method, where other tasks can be executed while waiting. Between two tasks the function $HasFinished()$ can be used to query the status in a non-blocking way. Second, a blocking waiting can be realised with the function $WaitUntilFinished()$. Compared to the polling method, this function has the advantage of smaller CPU load and shorter delays. Smart use of these functions enables the programmer to optimally exploit parallelism of the GPU.

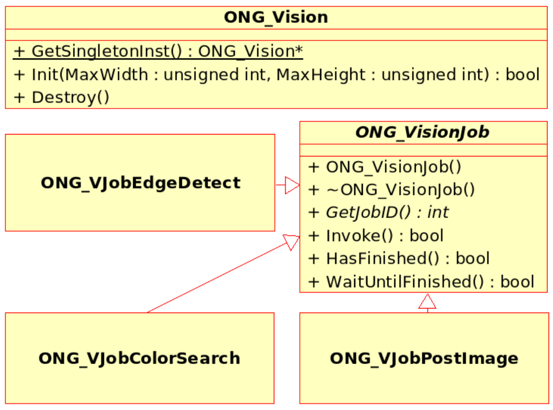

Fig. 3. Class diagram of the ONG library.

5 Samples

As an example for a successful GPU porting of an algorithm we will describe the implementation of the canny edge detection. The major difference for a CPU programmer is the vector based SIMD architecture.

To multiply a four dimensional vector with a 4×4 matrix, a high level language on a CPU consumes around 100 assembly instructions. For the same task the GPU needs only 4 instructions. The following four dimensional dot product:

```
C.x = A.x*B.x + A.y*B.y + A.z*B.z + A.w*A.w;
```

can be computed on the GPU in only one instruction:

```
DP4 C.x, A, B;
```

Especially for vector based mathematical computations many instructions can be saved. This advantage pays off in the implementation of low-level image processing algorithms, because colours, gradients and filter kernels can be represented easily with vectors and matrices.

Canny edge detection can easily be programmed in GLSL (see Figure 4). Memory accesses are realised via textures and, rather than in C/C++ array accesses, here computation of the two-dimensional indices is done by hardware. These hardware operations almost come for free, because they are implemented with dedicated transistors on the GPU. Further improvements can be achieved by utilizing hardware built in functions of the GPU like $length()$ and $normalize()$.

The pixel-shader shown in figure 4 will automatically be applied on all pixels in an area of the input texture by rendering a filled polygon over the screen. To render a polygon we use the immediate OpenGL render functions like $glBegin()$, $glEnd()$ and $glVertex3f()$.

We read the result of our computation back by $glReadPixels()$ which can be very slow on older hardware (without PCI-express) and driver releases but is usually fast on newer machines.

```
uniform sampler2DRect   img;

void main()
{
  const vec2 offset = {1.0, 0.0};
  vec2 gradient;

  gradient.x = texture2DRect(img, gl_TexCoord[0].xy + offset.xy).y
             - texture2DRect(img, gl_TexCoord[0].xy - offset.xy).y;
  gradient.y = texture2DRect(img, gl_TexCoord[0].xy + offset.yx).y
             - texture2DRect(img, gl_TexCoord[0].xy - offset.yx).y;
  gl_FragColor.xyz = vec3(length(gradient.xy) * 0.707107,
                          gradient.xy * 0.5 + 0.5);
}
```

Fig. 4. The canny GLSL shader program.

Due to limitations of the architecture, edge-detection and the following non-maximum suppression can not be performed in one pass. First the edge detection has to be executed and then non-maximum suppression on the stored gradients (see figures 5 and 6).

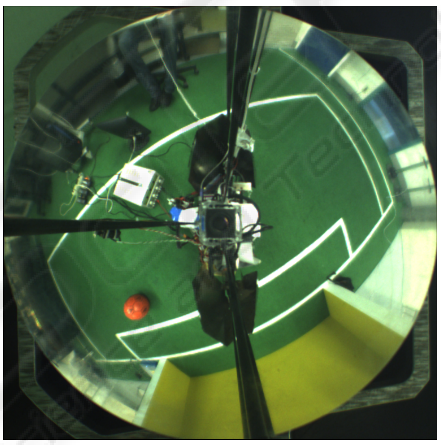

Fig. 5. Camera snapshot.

Since the GPU is optimized for floating point operations, filter kernels which are converted for CPU processing to integer arithmetic, can be executed without loss of performance in high precision on the GPU.

However, on the other hand there are some disadvantages of the SIMD architecture. Programming is more complex and control structures are strongly limited. For example, it is hardly possible to implement simple recursive algorithms or algorithms with nested loops.

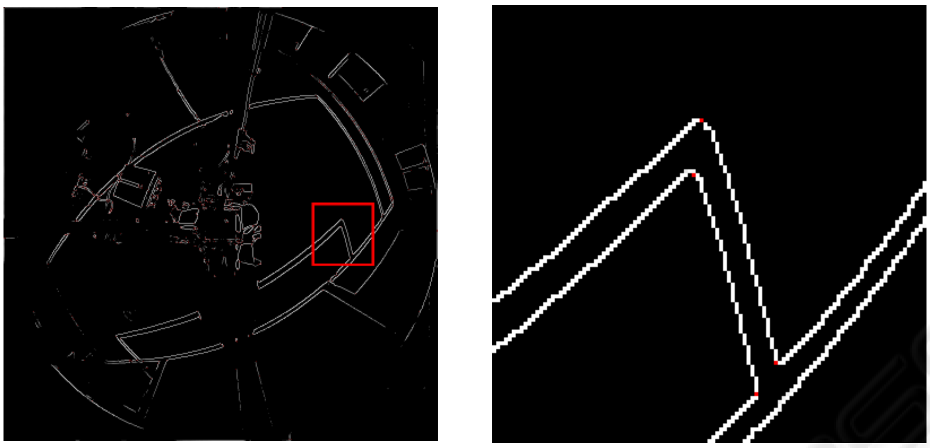

Fig. 6. Canny results. The right picture shows a closeup of the left picture.

6 Performance Results

In figure 7 the performance of our mobile system with GPU is compared to the same system without GPU. Including all the data transfers between CPU and GPU the GPU still is about a factor of 4 faster than the CPU. Thus, the edge detection alone can be done on the GPU with up to 50 frames per second, whereas on the CPU only about 12 frames per second are possible.

For the conversion of the Bayer pattern to a full RGB image (see [11]) we have to interpolate the missing colour channels from neighbour pixels. Because of the Bayer pattern, nearby pixels always require different treatment which makes cache area prediction difficult. Therefore the GPU gets problems using its full memory bandwidth. Together with the fact that the linear interpolations don't take enough time to hide memory latency, this explains why the GPU isn't even twice as fast as the CPU.

For Canny Edge Detection with non-maximum suppression it looks different. The calculations can be vectorised and so the GPU is about 5 times faster than the CPU which suffers from limited memory access bandwidth and too few registers.

Because of architecture limits requiring multiple passes, on the GPU the gradients computed by the Canny edge detection have to be saved for later reuse. Here, if the CPU with its limited memory bandwidth needs to store the gradients, the GPU is about 11 times faster. In our Robocup application Canny edge detection is followed by a corner detection, for which the gradients have to be stored. Thus, for the combined edge- and corner detection the GPU yields an overall speedup of about 4.

Optimization methods like region of interest can also be applied on the GPU as well as on the CPU. This can be achieved by rendering the polygon on which the pixel-shader will be executed only over a specific erea. The speedup for rectangular areas will be the same on both architectures. On the GPU it is also possible to render non rectangular areas without practical performance loss due to the shape complexity.

resolution:	1024x768 to 768 x 768
frame rate:	30 fps
colour depth:	24bit
GPU:	nVidia GeForce 6600
VRAM:	128MB
CPU:	Pentium M x86
CPU clock rate:	1600MHz
main memory:	512MB
operating system:	Linux Fedora Core 4
XServer:	7.1

Task	GPU(in ms)	CPU(in ms)
image transfer to GPU	1.82	–
image transfer to CPU	3.81	–
Bayer $\rightarrow$ RGB	6.84	12.9
RGB $\rightarrow$ HSV	2.63	14.91
Canny edge detection	4.52	21.04 / 53.42
total	19.62	48.85 / 81.23
speedup GPU vs. CPU		2.5 / 4.1

Fig. 7. Our mobile system (left) and performance measurements on it (right). The times on the CPU for the Canny edge detection are given without (left) and with (right) storing the gradients.

Figure 7 also shows that data transfer from the GPU to the CPU still is much slower than in the other direction although the PCI-express bus is symmetric. This has historical reasons and should be fixed soon by newer drivers and upcoming GPUs.

7 Conclusion

Due to its special design the GPU is very well suited for many low level image processing tasks. Measurements proof this. Depending on the task, the GPU allows much higher frame rates than the CPU. In realtime applications with fast moving scenes, this performance gain is crucial.

Acknowledgements

The authors would like to thank Franz Brümmer and the Robocup Team at HRW.

References

1. Realtime Vision for Mobile Robots on the GPU. www.robocup.hs-weingarten.de/gpu-vision
2. Harris, Mark. 2005. Mapping Computional Concepts to GPUs. In *GPU Gems 2*, edited by Randima Fernando, pp. 493-508. Addison Wesley.
3. Kilgariff, Emmett and Fernando Randima. 2005. The GeForce 6 Series GPU Architecture. In *GPU Gems 2*, edited by Randima Fernando, pp. 471-491. Addison Wesley.
4. General-Purpose Computation Using Graphics Hardware, www.gpgpu.org
5. Buck, Ian and Purcell, Tim. 2004. A Toolkit for Computation on GPUs. In *GPU Gems*, edited by Randima Fernando, pp. 621-636. Addison Wesley.
6. Fung, James. 2005. Computer Vision on the GPU. In *GPU Gems 2*, edited by Randima Fernando, pp. 649-666. Addison Wesley.
7. OpenGL - The Industry's Foundation for High Performance Graphics, www.opengl.org
8. OpenGL Shading Language Specification (version 1.20.8, September 7, 2006), www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf
9. OpenGL Extension Registry, www.opengl.org/registry

10. OpenGL Hardware Registry, www.delphi3d.net/hardware
11. RGB "'Bayer"' Colour and MicroLenses, www.siliconimaging.com/RGB\%20Bayer.htm