Benchmark of multi-core technologies through image segmentation algorithms using Cell processor, GPGPU architecture and SIMD techniques

Daniel Lélis Baggio (danielbaggio@gmail.com), Fernando Fernandes Neto (alphakiller_@msn.com), Thomás Dias (tcdias@gmail.com), Einstein do Nascimento Jr. (einsteinnjr@gmail.com) Mauro Eidi Vilela Assano(massano@br.ibm.com), Celso Hirata (hirata@ita.br) - Instituto Tecnológico de Aeronáutica - Brazil

***Abstract*— This article describes a performance comparison between different multi-core architectures while processing an image segmentation algorithm which is often used in computer vision and medical imaging. STI Cell processor, Graphic Processing Units (GPU) and SIMD architectures were benchmarked. Key bottlenecks of the application have been evaluated for each architecture showing which kind of approach best fits each one of them. GPU implementation was made through fragment pipeline in a data-parallel paradigm using NVidia's Cg language. Cell's implementation used SIMD, parallelism and double-buffering techniques. Mean segmentation times for a 256x256 RGB were: 3.3 ms in a CPU, 2.4 ms in a CPU with SIMD extensions, 1.0 ms in a GPU (8 pixel shaders), 0.87 ms in a PS3 Cell processor (6 SPE's) and 0.4 ms in another GPU (32 pixel shaders), which encourages the adoption of parallel approaches in this application. The main bottleneck has been identified as memory transfers. Source code has been made available at http://code.google.com/p/ps3hacking/.**

***Index Terms*— Benchmark, Cell Broadband Architecture, GPGPU, Segmentation**

I. Introduction

As microprocessors frequency increasing has slowed down, high performance computing community has sought for alternative architectures in order to find more processing power. Two interesting solutions that have increased performance on commodity processors are STI Cell and the recent use of Graphic Processing Units for general purpose computation, since fragment pipeline has become programmable. While Cell processor can be acquired for reasonable prices if bought along with a Play Station 3 – different from buying it in a Cell blade – GPUs have also reached lower cost levels, due to increasing demand for multimedia applications, thus creating an environment of low cost high processing personal processors.

Research applied to high performance computing can not only help applications that are desired to run at real time – such as medical video image processing – but also increase performance for supercomputing applications, such as quantum mechanical physics, weather forecasting, climate research, molecular modeling and physical simulations.

Previous work [1],[2],[3],[11] show that performance improvements have been obtained while using GPGPU, Cell and SIMD instructions.

The purpose of this paper is to compare processing times obtained using multi-core approaches for solving the image segmentation problem through a Laplacian kernel convolution and discuss the main advantages and difficulties of programming each architecture, showing what kind of approach is worth depending on number and size of segmented frames. In order to make benchmarks fully repeatable, source code of applications has been made available.

II. Architecture Description

A. Cell Broadband Engine

Cell, which is shorthand for Cell Broadband Engine Architecture (Cell BE), is a microprocessor jointly designed by Sony, Toshiba and IBM (STI) as the core of PlayStation3 gaming system. The great innovation of Cell is to combine a high performance general purpose 64-bit POWER Processing Element (PPE) with eight SIMD cores – the Synergistic Processing Elements (SPEs) – instead of using identical cooperating commodity processors. Each SPE has a dedicated local storage of 256KB and a Memory Flow Controller (MFC) [1]. A high level view of Cell BE first implementation is seen in Fig. 1. It must be noted that memory bandwidth with off-chip memory is 25.6 GB/s, while Local Storage (LS) and L2 Cache reach 51.2 GB/s.

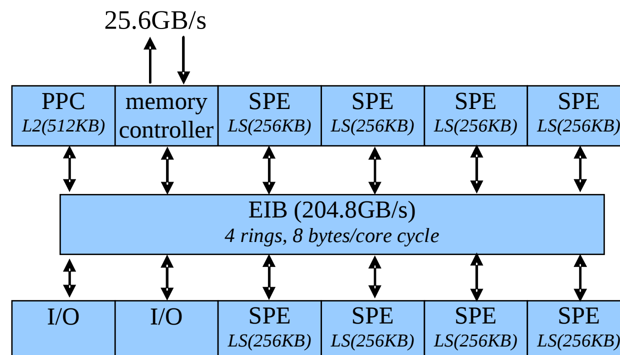

Fig. 1 – Cell Broadband Engine Processor Overview

Manuscript received August 31st, 2007
D. B. is with the Instituto Tecnológco de Aeronáutica (phone: 55-11-9533-64533 e-mail: danielbaggio@gmail.com).

The PPE is the core processor of the Cell BE and consists of a POWER Processing Unit connected to a 512KB L2 cache. It is a dual-issue, in-order processor with dual-thread support, designed to maximize the performance/area ratio as well as the performance/power ratio. The main purpose of this processor in the benchmark is for running the operating system (Debian) and coordinating the SPEs. It selects which one is supposed to run which task, as well as equally sharing the data for each thread. [2]

SPE consists of a Synergistic Processing Unit (SPU) and a Memory Flow Controller (MFC). It also has SIMD support and 256KB of dedicated local storage. Since it is a dual-issue, in-order machine, with an 128 entry, 128-bit register file, it can increase processing power in a wide range of applications. The 128-bit registers can process either integers or floating-point numbers, which must be in its dedicated local store, as well as the instructions it must perform. While SPUs process data, the MFC can be independently accessed and translate addresses for DMA transfers – a technique called double buffering, widely used in the application described in this paper. As the 128-bit registers can store 4 single-precision floating-point numbers of 32-bits, each SPU is capable of performing 25.6 GFLOPs, at 3.2 Ghz, which is the default Cell BE frequency. Eight SPEs can be accessed in parallel (while they are limited to 6 in PS3[19]), this adds up a total of 204.8 GFLOPs for single-precision floating-point numbers, which is highly desired for this specific application [2].

The Element Interconnect Bus (EIB) in the Cell is a high speed bus used for communication among the PPE, the SPEs, off-chip memory and the external I/O. It consists of 16B-wide data rings and one address bus operating at half the speed of the processor. Each unit of the EIB can simultaneously send and receive 16B of data every bus cycle and its theoretical peak data bandwidth, at 3.2GHz is amazing 204.8GB/s, what makes of this bus an excellent option for data streaming. [2]

B.General-Purpose Computation on Graphic Processing Units – GPGPU

The recent advent of Graphic Processor Units programmability and its increase in performance have made of GPUs a complete environment for general programming, yielding to what is so called General-Purpose computation on Graphic Processing Units (GPGPU).

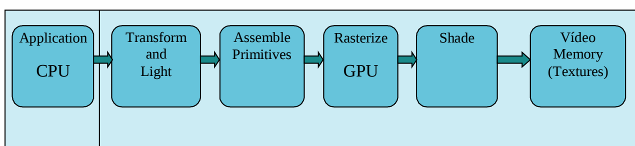

Fig. 2 – A simplified conventional graphics pipeline

Since 3D graphics have features that make them different from general computation domains, a whole pipeline has been developed in order to accelerate 3D environment rasterization. Fortunately, most of interactive 3D graphic application exhibit substantial parallelism, so an ad hoc pipeline has been created, as seen in figure Fig. 2, in which each stage is implemented as a separate piece of hardware.

The pipeline is fed with an array of vertexes which describe an object and outputs the image ready to go to the screen in a framebuffer that will be later used as input for another computation stage or will be drawn to a texture – which might or might not be displayed. The vertex processor transforms each three-dimensional vertex from object space into screen space, assembles vertexes into triangles and performs rotation, translation and scaling operations. This stage delivers two-dimensional triangles in screen space to the next one, called rasterization, in which triangle positions on the screen are determined, as well as their parameters are interpolated along the triangles. The output of rasterization is then delivered as fragments to the fragment processor, so that it can calculate the pixel color. This color is computed based on global texture - that may be in global memory - or in the color of the vertexes. The fragments are then assembled into a frame buffer that might be displayed to the screen [3].

According to [4], two steps in the pipeline are programmable: the vertex processor and the fragment processor. But they have limitations, which include:

- Access and modifications are only granted to their own data and the output format is fixed;
- They receive parameters in read-only mode;
- Limited number of assembly instructions: 1024 or 65536.

As GPU's were designed to render graphic workloads, programming for such units demands a paradigm shift. As pointed by [5], the first concept to be observed is that an array in general purpose programming is mapped to the texture concept. In CPUs, the native data layout are one-dimensional arrays – although two-dimensional arrays may be represented in a row-wise mapping. On the other hand, two-dimensional arrays are GPUs native data layout, although they are generally limited to 2048 or 4096 dimensions. So, the concept of data arrays in CPUs is mapped to the concept of texture coordinates in GPUs. The process of loading an array in GPUs is generally made through graphic libraries such as DirectX or OpenGL.

The second key concept in GPGPU programming [5] is that the computational kernel is mapped to the shader processing unit – also known as fragment processing unit. Nowadays GPUs have from 4 up to 128 fragment processing units, which result in hundreds of GFLOPs [6]. The fragment processing unit is the one used for general purpose programming. Often referred to as a group in the fragment pipeline, it consists of several processing units that behave as a vector processor of the size of the loaded textures. Since the order of the pixels processed in the pipeline cannot be programmed, it is assumed that all the data is processed in parallel without any data interdependence, in a paradigm known as data-parallel computing. The computational kernel, which may be a function of the form:

$$processed_y[i] = old_y[i] + constant * x[i]$$

(where i is an index in the output array, referring to a pixel) is sent to the fragment processor as a program, so that it can be independently run for each pixel available in the texture matrix. This kernel concept is mapped to GPUs in a pixel

shader program, which can be created with OpenGL shading language or NVidia's Cg (C for graphics). This program can be statically compiled or it can be compiled during execution, which is known as dynamic compilation. This approach is encouraged by NVidia [7]. Further details on how fragment program instructions are fetched are also found in the same book.

After the input texture has been assigned and the shader program has been loaded, the computation is made through the third GPGPU concept, which is drawing. It means that the only way to perform the computation in GPGPU's is outputting the result to another texture which might or might not be displayed - when the target is a framebuffer object. After drawing is made, all pixels in the original texture have been processed by the kernel computation, and the result may be used for another computation using the Ping Pong technique [5], for instance, or finishing the computation.

Since using graphic libraries API is not a trivial task for general programmers and this part of the process can be wrapped up in libraries, some alternatives have been proposed. One of them, developed by the Stanford University Graphics Lab is BrookGPU [8], a compiler and runtime implementation of the Brook stream programming language for modern GPUs. Another project that started academically is the Sh embedded metaprogramming language for programmable GPUs [9], originally developed by the University of Waterloo Computer Graphics Lab.

Another attempt to make GPGPU programming more straightforward was made by NVidia, in a technology known as CUDA (Compute Unified Device Architecture) [10]. This technology allows programmers to use an extension of the C language for a minimum learning curve and it also provides on-chip shared memory with faster access than DRAM, making applications less dependent on DRAM memory bandwidth. CUDA is compatible with NVidia's 8-Series, and CUDA SDK was made available in February 15th, 2007. AMD's competing GPGPU technology for ATI Radeon based series is called “Close to Metal”.

An important library focused in computer vision and image processing ported to GPUs is GPUCV. Previous benchmarks with GPUCV have shown speed gains of up to 18 times, in GeForce 7800 GTX (24 fragment processors) [4].

Since Cell and GPGPU are different architectures, some parameters must be watched in order to compare them. One of the most important is the number of floating-point operations per second, as shown in Table 1. As some applications also depend on how fast data is brought from off-chip memory to the processing units, peak bandwidth is also an interesting parameter to be compared.

C. Streaming SIMD Extensions

Single Instruction, Multiple Data (SIMD) techniques have been explored to achieve data level parallelism acting in a vector processor. This technique has appeared in processors through MMX technology, highly required for 3D applications [11]. After that, in 1999, Streaming SIMD Extensions (SSE) were introduced in Pentium III and later adopted by AMD. While MMX registers worked only with integers, SSE adds features to work with floating-point as well as 70 new instructions – 50 for SIMD floating-point numbers, 12 for SIMD integers and the remaining for cache. It has 8 registers of 128-bit size [12].

Table 1 – Key parameters for architecture comparison

	Estimated GFLOPS	**Clock Frequency (GHz)Speed**	**Off-Chip Memory Bandwidth (GB/s)**
Cell Processor	204.8	3.2	25.6
GeForce 8800 GTX	518.4	1.35	86.4
GeForce 7800 GTX	165	0.43	38.4
GeForce 6800 Ultra	54	0.4	35.2

One easy way to use SSE is through intrinsics, as suggested by [13]. Including the header <xmmintrin.h>, the programmer is able to access SSE instructions without managing registers by hand, while enabling the compiler to optimize instruction scheduling. Although SSE intrinsics aid programmers, the potentially fastest implementation would be directly accessing the 8 128-bit XMM registers.

III. Algorithm Description

According to [14], the convolution of an image with a Laplacian kernel approximates the second partial derivative of an image. This Laplacian image zero-crossing corresponds to points of maximal (or minimal) gradient magnitude, thus, representing good edge properties. Intensity changes at a given scale are best detected by finding the zero values of $\nabla^2 G(x,y)*I(x,y)$ for image I, where G(x, y) is a two-dimensional Gaussian distribution and ∇^2 is the Laplacian. In the implemented approach, a commonly used discrete approximation for the Laplacian filter was used [15]. It is defined as:

Table 2 – Convolution Kernel used for edge detection

-1	-1	-1
-1	8	-1
-1	-1	-1

It must be noted that this operation is evaluated over every pixel on the images. For each convolution, 9 operations are executed – 1 multiplication and 8 subtractions. As memory lookups and loop flow control are also needed, for each pixel, around 100 operations are made. This computation yields, for a 1024 x 1024, RGB image, around 3x10^8 operations (1024*1024*3*100), which is a high processing load for a standard CPU, if results are needed in real time - as for medical image segmentations.

IV. Benchmarked Algorithms

A. Standard CPU approach

CPU code uses a straightforward implementation of the algorithm. Firstly it loads a RGB bitmap into an array and then the convolution showed in Table 1 is applied over the image. Since operations are processed for each color channel, there is no need to insert zeros in each triple of values for alignment, as it is needed for SIMD and GPGPU. For details of the implementation, one should look for source code, which is available at [17].

B. SSE instructions used for segmentation

Optimizations introduced in the base CPU code were mainly adapting the data structure for correct SSE instructions alignment and adding intrinsics.

One loop over the original image structure was made adding one null value for each triple of RGB values. This correction was needed so that when the segmentation filter was applied each color value would be calculated with the corresponding neighbor pixel color. In case it was not made, red fields would be compared to green ones.

Besides using 128-bit registers, the main intrinsics used were: _mm_set_ps1, for setting constants, _mm_add_ps, _mm_mul_ps, _mm_sub_ps for respectively adding, multiplying and subtracting register values.

As operations with SSE occur over 4 floating-point numbers per instruction, the main filter loop was decreased by a factor of 3 – since one value is discarded for alignment reasons – yielding a good speed-up. Source code is available at http://code.google.com/p/ps3hacking/.

C. GPGPU Implementation

GPGPU code was implemented using standard OpenGL API, based on original code helloGPGPU, from Mark J. Harris [16](www.gpgpu.org). Hence the kernel convolution is made through a pixel shader programmed in NVidia's Cg (C for graphics).

Firstly, the loaded bitmap is transferred to an OpenGL Texture structure, and then it is displayed over a rectangular polygon (GL_QUAD structure), in a framebuffer object. After that, the resulting drawing is copied to another texture. This texture is the segmentation target and uses the GPGPU concept that an array for GPGPU is a texture. The segmentation kernel is then uploaded to the GPU, in which it is compiled and applied in parallel to many fragments simultaneously, during an orthographic drawing of the image. It shows the concept of binding a computational kernel to a fragment program. The result is then copied back to a texture, what makes feedback possible. For more details, one should look at [17].

D. Cell algorithm description

Since Cell makes memory management available for the programmer, its implementation is quite different from the previous ones. Two programs are made, one for the PPE and another for the SPE. The first one deals with SPE threads initialization as well as equally distributing data into them, which is made by dividing image rows to each thread. After initialization, SPE threads are started and each one acts over one sixth of the image – since only 6 SPEs are available in PS3. During SPE code, each one uses DMA Memory Flow Control calls so that three image rows are loaded. After that, the next row is required and the segmentation algorithm is run simultaneously. After the algorithm has completely processed the first three rows, it writes the results in a new row and move pointers to the recently loaded one, in an approach called Double Buffering. It must also be stated that all operations over image rows are made through SIMD instructions, increasing performance. The whole algorithm can be seen at [17].

V. Results and Analysis

Table 3 -CPU Segmentation Time for Different Image Sizes

Processor	256x256	512x512	1024x1024
Centrino Duo T7300	3.3 ms	12.3 ms	51 ms
Centrino Duo T7300 with SSE instructions	2.4 ms	9.6 ms	40 ms

A. Standard CPU code and CPU with SSE instructions

Table 3 shows processing times of filtering different sized images. The computer's processor used for benchmark was an Intel Core 2 Duo Centrino T7300, 2.0 GHz, with 2GB of RAM memory, running Linux Fedora Core 7. In order to count the time elapsed, system function *gettimeofday* was used (it is declared in sys/time.h). Code was compiled with GNU's g++ C++ compiler, with optimization flag –O3.

Besides benchmarking default CPU implementation, SSE extensions were also used. This code puts each color channel in the same vector instruction (as well as a spare channel, which is not used) making operations over 4 channels per instruction. The theoretical speedup of 3 was not observed, although code ran faster. This happens because not all of the operations are arithmetic, as there are several memory lookups and flow control operations.

Main advantages of CPU processing is that initialization time is null, since the CPU is already up and the results are stored in main memory.

B. Comparison of GPGPU

Table 4 shows results for what would be a typical usage of image segmentation and screen displaying for a an entire movie – such as a coronary exam in a medical application. The time considered for the application only reads images from CPU memory, transfers them to the GPU and runs the kernel program, displaying results in the framebuffer, without uploading them back to the CPU – what would be expected in case only viewing the results was enough.

GPGPU code was run over different GPUs in order to watch how much performance was gained as more pixel shaders were added. Benchmarked GPUs were Nvidia's GeForce 6 series: GeForce 6200 (4 pixel shader processors, 350MHz), GeForce 6600 GT (8 pixel shader processors, 500MHz) and GeForce 6800 Ultra (16 pixel shaders, 400MHz). A mobile version, GeForce8600M GT (32

processors) was also tested. Since image has to be retrieved from computer's main memory, the memory bandwidth should also be observed: GeForce 6200 (5.6 GB/s), GeForce 6600 GT (16.0 GB/s) and GeForce 6800 (35.2GB/s).

It is clear from results in Table 4 that the amount of time grows linearly with the number of pixels of the images (since each image is 4 times bigger as well as the processing time).

Table 4 – Download and segmentation mean time for 10,000 samples

GPU	256x256 pixels (ms)	512x512 pixels (ms)	1024x1024 pixels (ms)
GeForce 8600M GT	0.40	1.29	5.71
GeForce 6800	0.53	1.6	6.97
GeForce 6600	0.99	3.27	13.07
GeForce 6200	14.7	15.71	65.9

Running the segmentation thousands of times and taking the mean value may hide some important details about bottlenecks in the program. Separate tests were run for Nvidia GeForce 8600MGT and gave more insight about the application profile, as shown in Table 5. Firstly, 95 ms are required for the initialization of the context, through instruction *cgCreateContext()*. This function creates a Cg context object and returns its handle. According to [18], a Cg context is a container for Cg programs and all Cg programs must be added to a Cg context. Then, the best profile is chosen in 2 ms. Documentation of *cgGLGetLatestProfile*[18] shows that during this function, extensions are checked to determine the best profile which is supported by the current GPU, driver, and cgGL library combination. Then, it takes around 8 ms to execute function *cgGLLoadProgram*, which is responsible for preparing the program for binding.

Since preparing Cg Context is required only once per application, it is clear from Table 5 that the greatest bottlenecks of this program are memory transfers as it takes 5.6 ms for downloading the image to the GPU and 6.5 ms for uploading (although the segmented images are not required to be uploaded in case they are not going to be later recorded in the hard disk, as it might be enough to only display the results, depending on the application). This way, one can conlude that the speedup caused by using different models in Table 4 is due to their different bandwidths. Tables 6 and 7 show download and upload times for different image sizes.

As it can be seen in table 5, this application is extremely light in terms of computation, since the computational kernel only looks up and multiplay nine pixel values for each pixel, which is linear for the size of the image.

Table 5 – Major execution delays for GeForce 8600M GT

Task	Mean time (ms)
Create Cg Context	95
Chose best profile	2
Load fragment program	8
Download 1024x1024 image	5.6
Execute fragment program	0.06
Upload 1024x1024 image	6.5

Table 6 – Download Memory Transfer Delays

Image size (pixels)	Mean time (ms)
256x256	0.3
512x512	1.2
1024x1024	5.7

In order to download images to GPU, the function *glTexImage2D* was used, since the image was stored in framebuffer objects.

In case it was needed to record the processed images, they would have to be uploaded to the CPU memory. This is accomplished through the function *glReadPixels*, used to create Table 7.

Table 7 – Upload Memory Transfer Delays

Image size (pixels)	Mean time (ms)
256x256	0.5
512x512	1.8
1024x1024	6.5

C.Cell Benchmark

Table 8 – Cell (252x252) using 6 SPEs

Number of Samples	Total time (ms)	Mean time (ms)
16	30	1.9
128	128	1.0
256	240	0.93
10240	9010	0.87

Since this paper focuses on commodity processors, Cell benchmark was made in a PlayStation3 (PS3). Cell's environment was a Linux Debian 2.6.16 for Power PC64 with standard cell-sdk toolchain compilers. Table 8 shows results of running the algorithm for a different number of

samples. Since only 6 SPEs are available[19], the picture used was 252 pixels long so that it could be equally divided to the six of them. Each SPE received a 252 x 42, 4 channel image segment. The total memory size required for these segments is 42KB, which fits perfectly in local memory. It is clear from the results that SPEs suffer from initialization time, since mean time approaches 0.9 ms for 10 thousand samples and 1.9 for only 16 samples. In PPU code, the call for creating SPE threads is made through *spe_create_thread*, which is implemented by *libSPE* and is not visible to the programmers.

Another test, with 10240 samples for the same 252x252 image, using only 1 SPE was made and it totaled 53,621 ms, which is around 6 times higher than the code with 6 SPEs. This result shows a linear speedup provided by code parallelization.

Memory transfers were made through *mfc* calls and were copied back to main memory. One of the main advantages of this approach is that segmented results are ready to be stored, but, if the intention is to see the results, they will have to be downloaded to the framebuffer.

While generating code for Cell, optimization flags were tested with their respective impact on the execution performance. The most important one was "-fschedule-insns", being responsible for 70% to 80% of the performance increase (since original results were slower). As the code depends on several parallel computations made on a single variable - "temp1" on *cell_sc_spu.c*[17] - instructions should be scheduled in order to avoid SPU stalls caused by sequential operations over the same variable. As in the other architectures, "-O3" optimization flag was also used to get minor performance enhancements from the chip.

One of the bottlenecks for Cell processor is the initialization delay, which is caused by slow SPE context switch. The measured value for initialization has been 5ms to start 6 SPEs. In order to make computations on SPEs worth, a large number of frames or samples must be used, so that the average time gets reasonable.

Another key point of bottleneck is related to memory transfers to the memory controller in the element interconnect bus (EIB). Since the workload is light, SPEs can generate floating point results at a higher speed than the one at which data can be fed to them.

VI. Conclusions

Although some of the benchmarks were made in different architectures, with different clocks and memory access, it can be seen that image segmentation performance can be increased in all architectures, with software change (assuming a GPU or Cell processor is already available).

Simple use of SSE instructions has made a great increase in performance and small changes in code. As it does not require additional hardware, this solution is almost mandatory for programs running on standard CPUs.

Cell performance has shown a great increase, but programmability has still been an issue, since this enhancement depends not only in distributing tasks for SPEs but also explicitly managing memory access. Without proper software tools, programming becomes tougher although it is believed that programmers that deal with cache hierarchy of conventional processors will have little difficulty to learn Cell programming.

GPGPU also shows great results for using a commodity graphic processing unit in general programming problems. Since a graphical API was used, GPU programming was still not a trivial task. As a future study, Nvidia's CUDA API, mentioned in section II-B, should be used in order to check for performance increasings as well as easier programmability. Another difficulty in programming for multi-cores is that the problem must be correctly parallelized, which might not be trivial for different applications.

Results show that a performance increase of up to 9 times - GeForce 8600MGT over standard CPU approach for 512x512 image - can be obtained using relatively cheap GPU hardware. Since most of current personal computers are equipped with such hardware, GPU image libraries should be built in order to employ the spare performance available on GPUs. Another detail is that for tasks that are executed only once in a while, as might be the case of a single filtering in an image processing software, the CPU might be the best option, since it will not suffer from initialization delays, as happens for creating a context in GPUs or switching context to SPEs in Cell. It is also clear from the results that, as larger images or sequences of frames are used, this delay becomes insignificant, showing awesome speed increases.

References

[1] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. “The Potential of the Cell Processor for Scientific Computing.” In Proceedings of the 3rd conference on Computing frontiers, (2006), 9 – 20
[2] Cell broadband engine architecture and its first implementation. http://www-128.ibm.com/developerworks/power/library/pa-cellperf/
[3] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. “A Survey of General-Purpose Computation on Graphics Hardware” Computer Graphics Forum 26, 1 (2007), 80 – 113.
[4] J.P. Farrugia, P. Horain, E. Guehenneux, and Y. Alusse. “GPUCV: A Framework for Image Processing Acceleration with Graphics Processors.” Multimedia and Expo, 2006 IEEE International Conference On, (July, 2006), 585 – 588
[5] Göddeke, D.: GPGPU-Basic Math Tutorial, Ergebnisberichte des Instituts für Angewandte Mathematik, Nummer 300, FB Mathematik, Universität Dortmund, http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html, 2005
[6] E. Kilgariff and R. Fernando. “The GeForce 6 Series GPU Architecture.” In GPU Gems 2 Pharr M., (Ed.). Addison Wesley, Mar. 2005, ch. 30, 471–491.
[7] R. Fernando and M. J. Kilgard. “The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics”, (Ed.). Addison Wesley, Feb. 2003
[8] Stanford University Graphics Lab, “Brook gpu,” http://graphics.stanford.edu/projects/brookgpu/
[9] University of Waterloo, “Sh: A high-level metaprogramming language for modern GPUs,” http://libsh.org/
[10] NVidia CUDA Programming Guide Version 0.8, Feb. 2007
[11] A. Klimovitski. Using SSE and SSE2: Misconceptions and Reality. In Intel Developer Update Magazine, Mar. 2001
[12] B. Patwardhan. Introduction to the Streaming SIMD Extensions in the Pentium III, http://www.x86.org/articles/sse_pt1/simd1.htm
[13] Getting Started with SSE/SSE2 for the Intel Pentium 4 Processor, http://www.intel.com/cd/ids/developer/asmo-na/eng/popular/20240.htm
[14] D. Marr, and E. Hildreth. Theory of edge detection. In: Proceedings of the Royal Society of London. [S.l.: s.n.], 1980, 187–217.

[15] Michael Seul, Michale J. Sammon, Lawrence O'Gorman. Practical Algorithms for Image Analysis: Description, Examples, and Code, (Ed.). Cambridge University Press, (April 15, 2000) p.88

[16] General-Purpose Computation Using Graphics Hardware, “GPGPU”, http://www.gpgpu.org

[17] Paper source code, http://code.google.com/p/ps3hacking/

[18] Cg Documentation, http://developer.nvidia.com/object/cg_toolkit.html

[19] A. Buttari, P. Luszczek, J. Kurzak, J. Dongarra, G. Bosilca, “SCOP3, A Rough Guide to Scientific Computin On the PlayStation3”, Innovative Computing Laboratory, University of Tennessee Knoxville, (May 11, 2007)