Fisheye Lens Distortion Correction on Multicore and Hardware Accelerator Platforms

Konstantis Daloukas, Christos D. Antonopoulos, Nikolaos Bellas
Department of Computer and Communications Engineering,
University of Thessaly, Volos
Greece
{kodalouk, cda, nbellas}@uth.gr

Sek M. Chai
Motorola, Inc.
Schaumburg, IL
USA
sek.chai@motorola.com

***Abstract*—Wide-angle (fisheye) lenses are often used in virtual reality and computer vision applications to widen the field of view of conventional cameras. Those lenses, however, distort images. For most real-world applications the video stream needs to be transformed, at real-time (20 frames/sec or better), back to the natural-looking, central perspective space.**

This paper presents the implementation, optimization and characterization of a fisheye lens distortion correction application on three platforms: a conventional, homogeneous multicore processor by Intel, a heterogeneous multicore (Cell BE), and an FPGA implementing an automatically generated streaming accelerator. We evaluate the interaction of the application with those architectures using both high- and low-level performance metrics. In macroscopic terms, we find that todays mainstream conventional multicores are not effective in supporting real-time distortion correction, at least not with the currently commercially available core counts. Architectures, such as the Cell BE and FPGAs, offer the necessary computational power and scalability, at the expense of significantly higher development effort. Among these three platforms, only the FPGA and a fully optimized version of the code running on the Cell processor can provide real-time processing speed. In general, FPGAs meet the expectations of performance, flexibility, and low overhead. General purpose multicores are, on the other hand, much easier to program.

***Keywords*-Cell, FPGA, Image Warping, Performance Evaluation**

I. INTRODUCTION

Difficulties in scaling single-thread performance and limiting the power envelope in high performance processors, has forced CPU vendors to introduce general purpose multi-core units in a single die. Moreover, there is a growing trend in the High Performance and Desktop computing communities to include accelerators to speed up time consuming, number crunching application kernels. Reconfigurable logic, such as FPGAs, vector processors such as the Synergistic Processing Elements (SPEs) in Cell processor, and Graphics Processing Units (GPUs) have been shown to speed up applications in multimedia, graphics, data mining, scientific computing, etc. by orders of magnitude, compared to conventional multi-cores [10] [22].

There is little systematic research on how accelerators based on different computing substrates, such as multi-cores, vector accelerators, and reconfigurable devices compare in terms of performance, memory behavior, and ease of programming. This paper attempts to advance our understanding in these issues by characterizing an important image warping application - fisheye lens distortion correction - on three contemporary platforms, namely an x86 Chip Multiprocessor (CMP), the Cell processor, and a Virtex-4 FPGA (III).

Fisheye lenses allow imaging a large sector of the surrounding space instantaneously (II). While ordinary rectilinear lenses map incoming light rays to a planar photosensitive surface, fisheye lenses map them to a spherical surface, which is capable of a much wider field of view (FoV). It is possible, and in fact very common, for fisheye lenses to encompass a FoV of 180^o. Such hemispherical images have been traditionally used for applications such as consumer digital imaging and video capture, video surveillance [18], robot navigation [14], content creation for immersive environments and virtual reality [27], photography [29], astronomy, etc. Fisheye lens distortion correction is an image warping application which transforms the distorted images back to the natural-looking, central perspective space (Fig. 1).

This paper explores how the inherent parallelism of the wide-angle lens distortion correction algorithm is exploited on different computational fabrics to achieve real time functionality for megapixel input frames (IV). It also presents a detailed characterization of macroscopic performance (IV-A) and lower-level metrics (IV-B). Although the algorithm has a high degree of data level parallelism at multiple levels of granularity, the exploitation of this parallelism is not trivial due to complex memory access patterns.

Some of the most important findings of the paper are the following: although the architecture of cache-based, general purpose multi-core systems has been described as being mismatched to streaming workloads, due to lack of spatial locality of streaming data references, our characterization shows that this is not necessarily the case. For streaming imaging applications that require substantial processing per pixel, such multi-core processors perform at least as good as the Cell processor per executing thread. Both platforms achieve speed up which is linear to the number of executing threads ($8x$ and $4x$ for the Cell and Core 2 Quad, respectively).

Moreover, processors that rely on spatial computing to "spread out" parallel tasks (Cell and FPGA), require placement of pixel data close to the execution cores to meet performance

978-1-4244-6443-2/10/$26.00 ©2010 IEEE

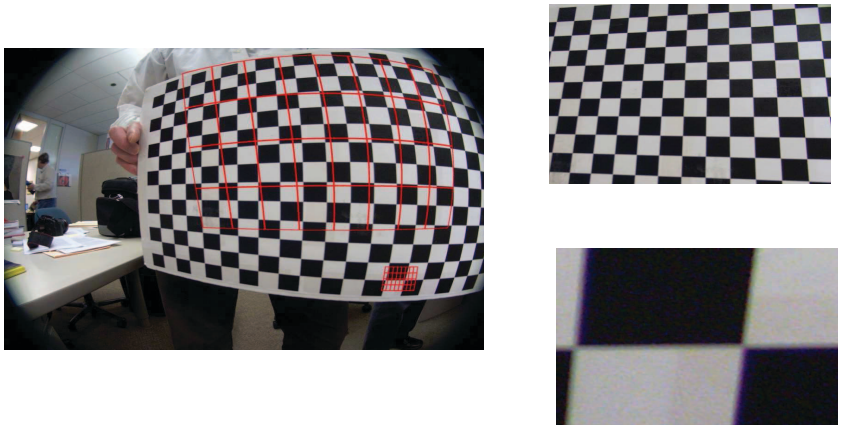


Fig. 1. The wide-angle lens distortion correction algorithm for two cases for field of view FoV = 60 ^{o} and FoV = 8^o. The output images are VGA (640x480). The lower FoV results into a zoomed output image.

requirements. By placing pixel data in the Local Stores of the SPEs and the on-chip SRAMs of FPGAs, the distortion correction application becomes compute-bound, rather than memory bound, and can meet the bandwidth requirements of multiple independent threads. This pixel placement is more critical for FPGAs which cannot rely on high-speed buses and memory controllers to feed their computation units.

In addition to performance, development time is recognized as an important component of overall effectiveness of a target platform. Although FPGA devices have the highest development time, the effort to develop and optimize the application on the Cell processor was actually comparable to that on the FPGA device, especially when using an architectural synthesis tool [5] to map the application to the reconfigurable fabric.

II. FISHEYE LENS DISTORTION CORRECTION ALGORITHM

The stereoscopic geometry of wide-angle photography does not comply with the conventional central perspective projection shown in Fig. 2a. The latter is based on the premise that the incidence angle of an incoming ray from an object point is equal to the angle between the ray and the optical axis. Object points with incidence angle close to 90^o would be projected to a point at infinite distance from the principle point, thus limiting the FoV to angles close to the optical axis.

The wide-angle projection model is based on the principle that the incidence angle is proportional to the distance between the image point and the central point (Fig. 2b). The incoming rays are refracted closer to the optical axis, thus expanding the FoV.

In order to associate the coordinates (*i,j*) of a point at the 2D central perspective image space to the coordinates (*x,y*) of the corresponding point at the wide-angle space (inverse mapping), one has to first compute the coordinates (X_c, Y_c, Z_c) of the projection of the (*i,j*) point to the 3D camera coordinate system by applying a rotation matrix:

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \times \begin{bmatrix} i \\ j \\ 1 \end{bmatrix}$$

After some algebraic transformations [25], the equations that describe the projection on the image plane when using wide-angle lens are:

$$x = \frac{\frac{2R}{\pi} \cdot \arctan(\frac{\sqrt{X_c^2+Y_c^2}}{Z_c})}{\sqrt{(\frac{Y_c}{X_c})^2+1}} + d_x + x_h \quad (1a)$$

$$y = \frac{\frac{2R}{\pi} \cdot \arctan(\frac{\sqrt{X_c^2+Y_c^2}}{Z_c})}{\sqrt{(\frac{Y_c}{X_c})^2+1}} + d_y + y_h \quad (1b)$$

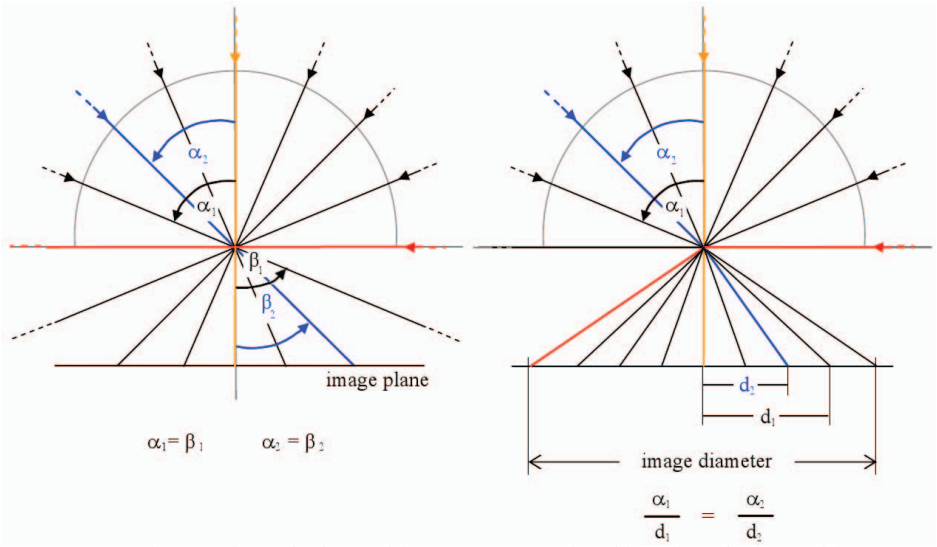
where (X_c, Y_c, Z_c) are object point coordinates on the 3D camera coordinate system, d_x, d_y are lens-distortion parameters, x_h, y_h are the coordinates of the principle point and R is the image radius.

Note that equations (1) produce a fractional pair of coordinates at the wide-angle plane, and the pixel value at that point has to be interpolated based on the values of the pixels at neighboring integer positions.

Bicubic interpolation is a robust, yet computationally expensive technique used to approximate intermediate points of a continuous event given the interpolation nodes, or sample points [16]. Although other techniques such as nearest neighbor or bilinear are simpler and more widely used in hardware implementations, the high PSNR[1] requirements of the fisheye correction module make this the method of choice. The bicubic interpolation method uses cubic sampled functions to approximate an intermediate value based on the fundamental property that the sample function *is* equal to the interpolation function in the sample points.

As a last step, at an extra computational cost, we apply

[1]Peak Signal to Noise Ratio is frequently used to measure signal quality in images.



(a) Central Perspective projection model (b) Wide-angle projection model

Fig. 2. Projection model of fisheye lens.

1: {**Input**: The frames (in the wide-angle space) to be corrected}
2: {**Output**: The corrected frames (in the perspective space)}
3: **for all** frames **do**
4: **for all** pixels in the output frame **do**
5: Compute the corresponding fractional position in the input frame (`InverseMapping()`)
6: Interpolate the pixel value at that fractional position (`BicubicInterpolation()`)
7: **end for**
8: Apply a 2-D low-pass filter to resize the output frame (`LPF()`)
9: **end for**

Fig. 3. High-level pseudocode of the fisheye lens distortion correction algorithm.

a 5-tap vertical and a 5-tap horizontal low pass filter on the corrected image in order to downscale it to a VGA (640x480) output resolution[2]. The low pass filter has the additional positive effect of eliminating any potential high frequency artifact noise on the image. The pseudo-code in Fig. 3 outlines the algorithm. The inverse mapping and two-dimensional bicubic interpolation flows are depicted in more detail in Fig. 4.

III. INTRODUCTION TO TARGET PLATFORMS

A. Intel Core 2 Quad

The Intel Core 2 Quad Q9300 is a representative implementation of mainstream, homogeneous multicore systems. The processor is clocked at 2.5 GHz and supports a 1.3 GHz FSB (front side bus). It is organized as two independent dual core processor blocks packaged together. Each of the dual core blocks integrates a 3MB L2 cache (12 way set associative, 64 bytes cache line), shared between the two cores of the block. Moreover, each core accesses a 64KB private L1 cache (32KB data + 32KB code, 8-way set associative, 64 bytes cache line). If the two dual core blocks have to communicate, they do so through the FSB. The processor has a thermal design power of 95W and supports the SSE 4.1 vector instructions set. The system we used for our experimental evaluation is equipped with 2GB dual-channel, DDR2 RAM, clocked at 667 MHz.

[2]The fisheye lens distortion correction algorithm is used in the context of a video conferencing system.

B. Cell Broadband Engine (CBE)

The Cell BE is an heterogeneous multicore processor. It integrates 8 Synergistic Processing Element cores (SPEs) and a separate 2-way SMT PowerPC core (PPE) [15]. These 9 cores, the main memory and the I/O interfaces are connected by an on-chip network, the Element Interconnect Bus (EIB). The processor is clocked at 3.2 GHz. Each SPE is organized as an 128-bit wide SIMD computational engine (Synergistic Processing Unit - SPU) and a Memory Flow Controller (MFC). There is a single 128x128 bit register file per SPE. Each SPE accesses a private, 256KB local storage (LS). The LS has bandwidth and latency characteristics similar to those of an L1 cache, however its content is explicitly software-controlled. It is shared by both program code and data. SPEs can not directly access the main memory. They can, instead, issue asynchronous DMA requests to transfer data between main memory and the LS, or between LSs of different SPEs. Up to 2 SIMD instructions can be issued per cycle - although specific instructions on each issue slot - resulting to a maximum theoretical performance of 204.8 Gflops for single-precision and 14.63 Gflops for double-precision floating-point operations. The typical power consumption envelope of a CBE processor is 60-80W.

The system we used for our experimental evaluation is an IBM QS20 blade, equipped with 1 GB of external DRAM.

C. Reconfigurable Logic

Compared to the fixed hardware of the Core 2 Quad and Cell architectures, reconfigurable devices (FPGAs) are essentially

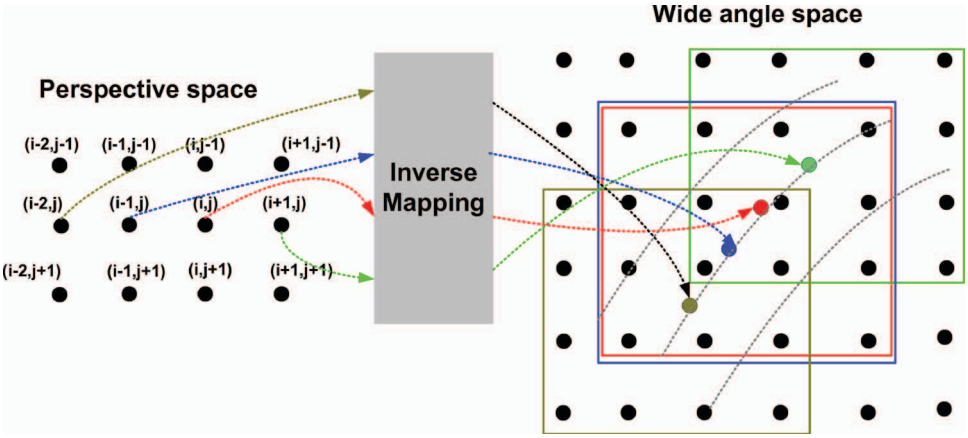


Fig. 4. Inverse mapping is used to convert the coordinates from the perspective space back to the wide-angle space. A 4x4 pixel neighborhood around the fractional points on the distorted space is used to perform bicubic interpolation and compute the pixel values at the fractional points.

high density arrays of uncommitted logic blocks that are configured post-fabrication [7]. The functionality of FPGAs is determined through configuration bits which are used to specify the functionality of the configurable logic blocks and the routing channels between them. Modern FPGAs also contain "islands" of hard intellectual property (IP) logic such as general purpose microprocessors, slices of DSP logic, and on-chip SRAMs.

Reconfigurable devices offer the highest degree of flexibility in tailoring the architecture to match the application, since they essentially emulate the functionality of a custom chip, i.e. ASIC (Application Specific Instruction Set). FPGAs avoid the traditional ISA-based von-Neumann architecture, followed by CPUs and the Cell processor, and can trade-off computing resources and performance by selecting the appropriate level of parallelism to implement an algorithm. Since reconfigurable logic is more efficient in implementing specific applications than multi-core CPUs, it enjoys higher power efficiency than any general purpose computing substrate.

The main drawbacks of FPGAs are twofold: first, the FPGAs are primarily programmed using Hardware Description Languages (VHDL or Verilog), which is a time-consuming and labor-intensive task, and requires deep knowledge of low-level hardware details. Although there has been a growing trend to program FPGA applications using high level languages, such as C-like languages [11], [19] and [23], most FPGA developers continue to use VHDL or Verilog to map their applications into the reconfigurable fabric.

Second, the achievable clock frequency in reconfigurable devices is lower (by almost an order of magnitude) compared with the full custom design of high performance processors. In fact, almost all FPGA designs operate in a clock frequency less than 200 MHz, despite the aggressive technology scaling of FPGA devices[3].

We use the Virtex-4 LX80 FPGA to implement the distortion correction hardware module. The LX80 device includes 80,460 logic cells, 200 on-chip SRAMs (18 Kbit each) and 80 DSP slices [2]. The hardware module is part of an embedded System On Chip (SoC), which also includes a Microblaze processor, a multi-port memory controller to provide high-bandwidth access to external SDRAM memory, and a variety of peripheral units. The FPGA device operates using a single clock at 62.5 MHz.

Using an internally developed architectural synthesis toolset and programming methodology, we generated the FPGA module without using a hardware description language. Proteus, our CAD tool [5], produces hardware accelerators that follow the streaming architectural paradigm [3]. This approach produces several independent load/store units (called stream interface units, SMIFs) used to prefetch data from wide, slow memories and turn it into narrow, high-speed streams of vector elements. It also generates the data path used to execute the program, which is expressed using an assembly-like streaming Data Flow Graph (sDFG).

Design automation allows us to turn FPGA programming from gate-level to algorithm-level and quickly convert the sDFG for our application (approximately 800 lines of code) into very efficient, synthesizable Verilog (approximately 100,000 lines of code).

IV. ALGORITHM OPTIMIZATIONS FOR PARALLEL EXECUTION

This section outlines the restructuring and optimizations of the original code in order to exploit the diverse parallel architectures of Core 2 Quad, Cell, and FPGA. Some optimizations, especially the higher level ones, are common to all platforms. Others are only fit for specific architectures.

A. High-level Optimizations

An important observation from the algorithmic analysis of section II is that the fractional pixel coordinates follow a complicated non-linear pattern (Fig. 4). Although the trace is not data dependent, and thus can be theoretically pre-computed, the complex memory access pattern deems aggressive DMA prefetching impractical.

[3]Next generation Virtex-6 FPGAs from Xilinx will be fabricated at 40nm CMOS technology.

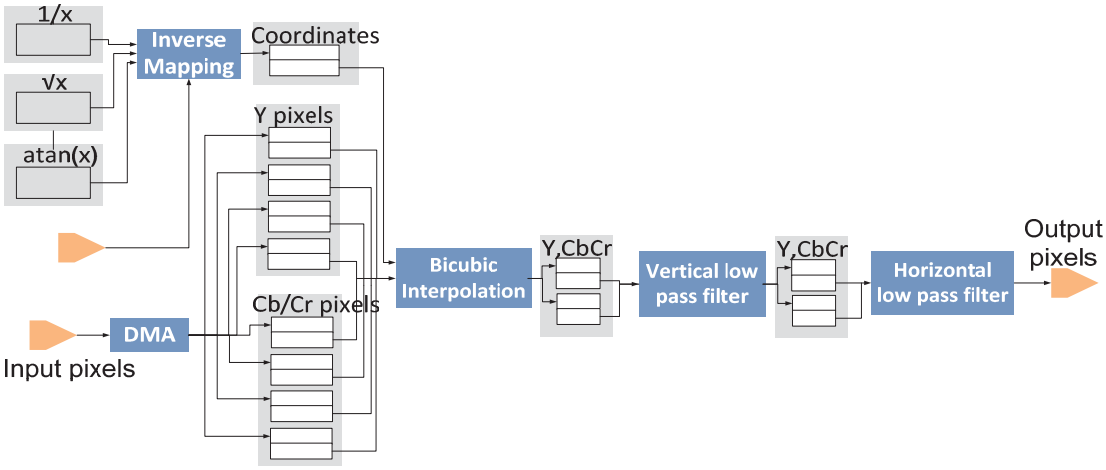


Fig. 5. Block diagram of the fisheye lens distortion correction pipeline implemented in reconfigurable logic.

We alleviate this problem by applying 2D tiling in each frame, a technique used by optimizing compilers to improve cache hit rate. We partition the output frame in blocks of equal size, and produce pixels block by block, by assigning one block to one thread (Core 2 Quad) or SPE (Cell). By tiling computations to exploit reuse at the block level, we also facilitate data distribution to SPE Local Stores in the Cell processor, and to the on-chip SRAMs of the FPGA, and we improve locality on the cache memory hierarchy of the Core 2 Quad architecture. Tiling allows us to store the working set within a small and constant latency from the computational units, instead of in remote (and possibly off-chip) memories.

Moreover, organizing computations around pixel tiles facilitates task-level pipelining and allows multiple tiles to be processed in the computational pipeline at any moment. In this scheme, each pipeline stage is dedicated to a single transformation so that successive tiles are processed simultaneously. For example, while all pixels of tile N are being processed in the bicubic interpolation stage, the tile N+1 is in the inverse mapping stage. We apply pipelining only to the FPGA implementation, which can exploit customization of the different pipeline stages to the task they execute. Pipelining is also possible for fixed architectures such as the Core 2 Quad and the Cell, however the application offers enough data-parallelism to exploit the 4 and 8 respectively execution contexts of these processors. Fig. 5 shows the pipelined block diagram of the streaming accelerator for fisheye lens distortion correction as implemented in the Virtex-4 FPGA.

B. Low-Level Optimizations

The fisheye lens distortion correction algorithm has abundant data level parallelism which can be exploited by the SIMD extensions of Core 2 Quad and Cell. Most calculations shown in Fig. 3 are enclosed in doubly-nested loops. The outer loop (pixel scan) first computes the fractional coordinates of each pixel in the tile (inverse mapping) and then applies bicubic interpolation within a nested loop of three iterations, one iteration for each of the color components (RGB). Likewise, two subsequent outer loops are used for the vertical and horizontal low-pass filtering, each enclosing a second-level nested loop with three iterations.

We utilize the SIMD capability of Core 2 Quad and SPEs by vectorizing four FP operations in inverse mapping, bicubic interpolation and low pass filtering. The implicit loop unrolling due to vectorization has the positive side-effect of reducing the backward branches of the outer loop by a factor of 4. An additional explicit $3x$ unrolling of the inner loops furthers the positive effects of branch elimination and increases the potential for efficient instruction scheduling. This is important for SPEs, in which mispredicted branches incur a large penalty. The SIMDization and loop unrolling provides a cumulative speed up of $12.4x$ for Cell, and $1.7x$ for the Core 2 Quad.

The loop unrolling optimization produced a large number of instructions that could be scheduled in parallel in the dual-issue pipeline. However, experimental results showed that the compiler was too conservative on rescheduling independent instructions. In order to reduce the pipeline stalls and produce a faster executable, we manually schedule the instructions at the source code level. We interleave the instructions that access the memory with computational operations, thus enabling the compiler to schedule them more efficiently and eliminating most of the pipeline stalls.

Bicubic interpolation accesses memory addresses that are not optimally aligned for vector loading operations (Fig. 4). The aforementioned implicit loop unrolling requires each row of the 4x4 window to be stored in a vector register, so the compiler inserts extra instructions in order to move the scalar data to the preferred slot. If vector registers are loaded row-wise, data dependencies from previous loads stall future loads on the same register, making the vector register a point of contention. We eliminate most of these stalls by reversing the order of storing in the vector registers. We load the registers

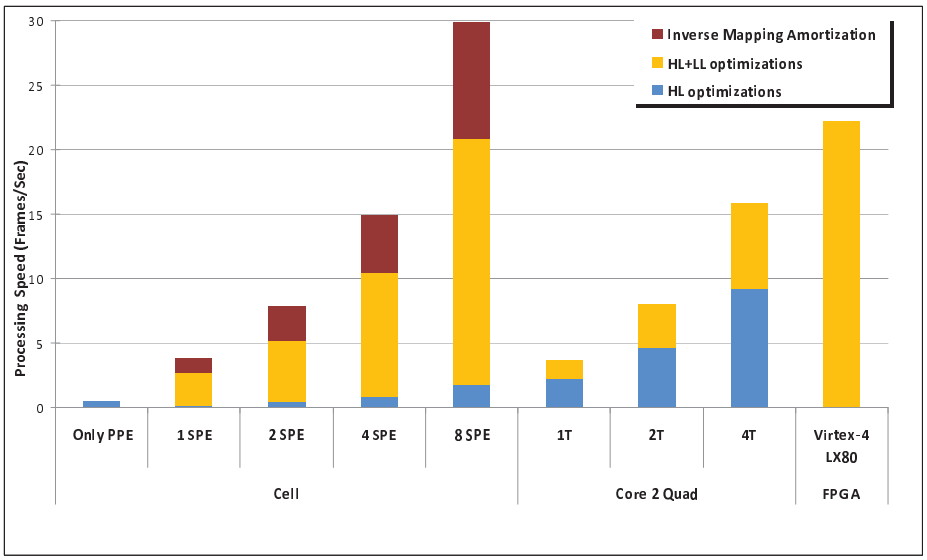


Fig. 6. Execution time and performance scalability. HL and LL stand for high-level and low-level optimizations respectively, whereas IMA stands for inverse mapping amortization.

column-wise instead of the intuitive row-wise order. This modification spreads out the instructions that access the same register, allowing enough time to load a pixel into a register, before loading the next one on the same register.

Moreover, the function that performs bicubic interpolation includes a conditional statement that checks whether a given location is inside the frame boundaries. After applying the SIMD optimization, we need to extract the coordinates of each point from the corresponding vector registers in order to evaluate these statements. We face a problem similar to that of unaligned loads, since we need to extract the individual coordinates, calculate the value of the conditional, and then execute or bypass the instruction in the body of the conditional statement. In order to eliminate the additional stalls that are inserted due to the consecutive extraction operations from a vector register, we vectorize the calculation of the conditionals, move them at the beginning of the outer loop, and extract the conditional values enough cycles before they are needed. The aforementioned low-level optimizations are described in more detail in [8].

A final optimization step for the Cell processor is to move the inverse mapping task to the PowerPC (PPE) processor, instead of the SPEs. As aforementioned, the correspondence of input versus output pixel coordinates depends solely on the region of interest (ROI) and the field of view (FoV), but not on pixel values. These two parameters can be changed interactively at run-time, however this occurs infrequently, if at all. As a result, the cost of inverse mapping can be amortized, if it is computed once and reused across multiple frames. This is achieved at the expense of storage space: the size of the resulting data structure is 4.8 MB, since it contains 1280x960 pairs of single-precision floating points. However, a data structure of that size cannot be accommodated in the local store of SPEs. In section V.B we evaluate this option in detail.

The FPGA implementation of Fig. 5 exploits the flexibility of the reconfigurable fabric by scheduling a large number of sDFG operations (around 400 in all pipeline stages) in each cycle using the modulo scheduler of Proteus. By placing incoming pixel data, and inter-stage intermediate results in the on-chip SRAMs of the reconfigurable device, the architecture keeps the wide data path fully utilized, and eliminates stalls due to memory latency. The intermediate on-chip buffers of Fig. 5 play the role of Local Store buffers in the Cell processor. The fisheye distortion correction module for the FPGA was developed using the Proteus CAD tool [5] and is described in more detail in [6].

We evaluated the execution time of the application with a FoV varying from 1.0^o to 60.0^o and for all possible ROIs on the input frame, and we found the execution time to be insensitive to these parameters. This was expected, since the size and resolution of the output image are fixed, and the amount of computation per output image pixel is not dependent on the input data and parameters. For the rest of the paper, we assume that the FoV is 40.0^o.

V. PERFORMANCE EVALUATION OF THE FISHEYE LENS DISTORTION CORRECTION ALGORITHM

In this section, we evaluate the performance of fisheye lens distortion correction on the three platforms we described in section III. All results are obtained from executing the application on real hardware, rather than on simulators.

The application was compiled on the Core 2 Quad using both Intel's icc compiler and gcc, with the compiler optimization flags that resulted to the lowest execution times. The performance of executables produced by icc proved slightly

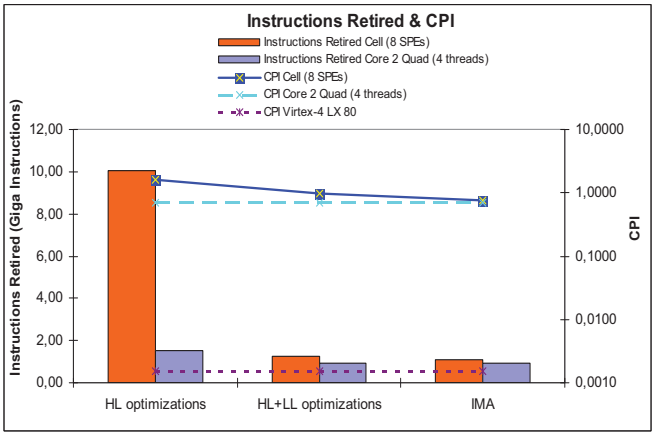


Fig. 7. Total number of retired instructions and CPI for the fastest implementation on each platform.

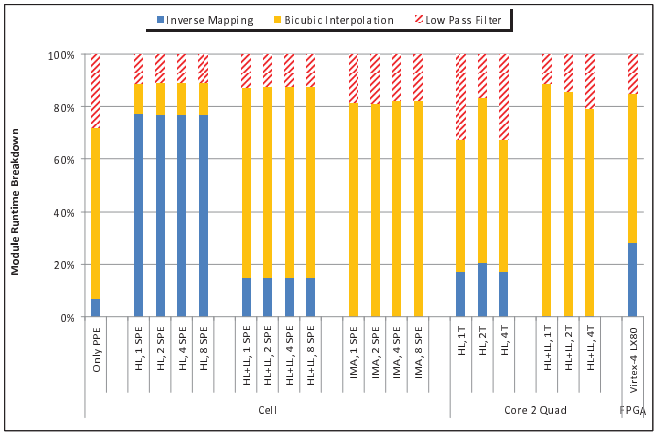


Fig. 8. Runtime breakdown for each function.

higher, so we only report these results. The code on the Intel-based platform has been parallelized using POSIX threads. Performance data have been collected by Intel VTune performance analyzer [13] and Intel Thread Profiler [12].

The application was compiled on the Cell processor using both gcc (version 4.2.1) and xlc compiler (version 9.0). The performance of the executables produced by gcc was higher, so we only report results using gcc. Low-level performance data have been collected by the Cell Performance Counter tool (CPC) [1] which is used for setting up and monitoring the hardware performance counters in the Cell processor. These counters allow the user to quantitatively evaluate interactions at the hardware / software boundary.

A. Performance and Scalability Analysis

Fig. 6 illustrates the performance of the application in terms of processed frames/sec in each platform after each optimization, i.e. HL for high-level optimizations only, HL+LL for high- and low-level optimizations, and finally IMA (inverse mapping amortization) when inverse mapping is executed once and the fractional coordinates are stored in memory and reused.

As a first observation, the speed up is proportional to the number of execution contexts available by the underlying platforms (i.e close to $8x$ and $4x$ for Cell and Core 2 Quad, respectively). This is expected, since there is no dependency between threads in this application. Moreover, the memory and bus hierarchies are able to keep the data paths close to full utilization, i.e. the application is compute-bound for all cases we measured. Assuming that we require real-time processing of at least 20 frames/sec (for human viewing), we observe that only the Cell processor and the FPGA can deliver it. The scalability results, combined with the low-level performance analysis discussed in the following subsection allow us to speculate that, should the Core 2 Quad processor be equipped with 8 cores, it would also probably be capable of achieving real-time performance. In all cases, reconfigurable computing shows its advantages over von Neuman-based CPUs, since the FPGA is $37.5x$ faster per Hz of execution than the Cell processor using all eight SPEs, and $56x$ faster per Hz than the Core 2 Quad processor using all four threads.

Exploiting fine-level parallelism with vectorization and loop unrolling provided $12.4x$ speed up for Cell and $1.7x$ speed up for Core 2 Quad, compared with the HL optimizations only. This is a testament of careful source code rewriting (most probably, manual) needed to optimize SPE execution. The lack of a dynamic branch prediction mechanism in hardware, the lack of automatic or compiler assisted SIMDization, as well as the overly conservative instruction scheduling by the SPE compiler, place the burden on the programmer to produce optimized code.

Finally, inverse mapping amortization proved to be beneficial only for the Cell processor and provided an extra $1.43x$ speed up with respect to the HL+LL optimizations. As far as the Core 2 Quad processor is concerned, after applying the LL optimizations the compiler produced very optimized, inlined code for inverse mapping function, thus making its contribution to the total execution time negligible.

Fig. 7 depicts the total number of retired instructions and the effective CPI for the three platforms when all available resources are used. Note that the CPI of the FPGA at around 0.0015 is approximately 500 times smaller than the CPI of the two CPUs. The three platforms are based on different ISAs, thus a direct CPI comparison among them is not valid. Nevertheless, this number is a good indication of the superior resource utilization offered by reconfigurable devices.

In order to better understand the reason why SIMD optimizations have such a different effect on the Core 2 Quad and Cell platforms, we break down the function execution times in Fig. 8. The SPEs in Cell processor spend approximately 75% of their time executing inverse mapping in the HL optimization case, i.e. before any low-level optimizations, whereas the Core 2 Quad spends only 18%. These percentages drop to 15% and 0.3% (due to inlining and compiler optimizations) respectively after the LL optimizations are applied.

This runtime imbalance is mainly due to the superior

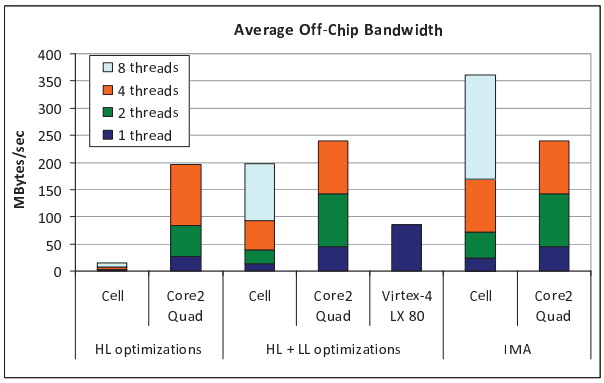


Fig. 9. Average off-chip bandwidth requirements.

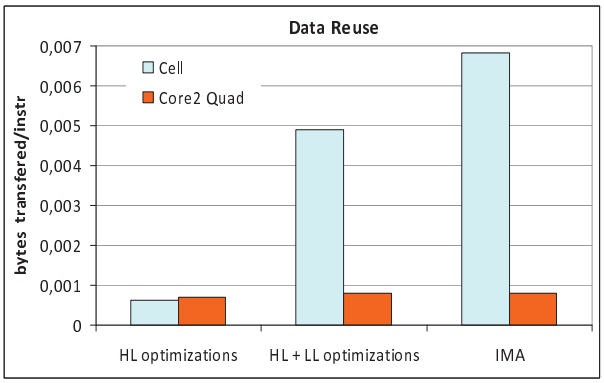


Fig. 10. Data transfers (bytes) from off-chip memory per instruction.The rates reported have been observed when executing with the maximum concurrency supported by each platform (4 threads on the Core 2 Quad, 8 SPEs on the Cell BE). The results are indistinguishable when less execution contexts are used.

performance of the x86 floating point unit and the capability to execute the elementary functions of eq. 1 in hardware, rather than with software libraries, as in Cell. In that case, vector optimizations are critical in improving the total execution time.

Note that in the pipelined FPGA module, the execution time per frame is equal to the longest executing task, which is the bicubic interpolation. In this case, Fig. 8 implies that the low pass filters (both vertical and horizontal) and the inverse mapping finish earlier and have to spin-wait before they start with the next tile. The limiting factor in exploiting additional ILP in the bicubic interpolation stage, in order to reduce its latency, is the number of output ports in the pipeline SRAMs between stage 1 and stage 2 of the FPGA.

B. Memory Performance

A key performance limiting factor in many streaming applications is the off-chip bandwidth requirements. Fig. 9 depicts the amount of data transferred per second from main memory to the chip for the three target architectures. The amount of data transfers per frame is almost fully predictable. Approximately 4.98 MBytes of input data need to be fetched, in order to generate an output frame of 900 Kbytes which has to be stored on disk or presented on screen. Inverse mapping amortization across different frames may generate additional traffic. If the combined capacity of outer-level caches - i.e. the L2 for Core 2 Quad and all LSs for Cell BE - is not sufficient to fully accommodate the working set for the computation of a frame, the fractional coordinates, namely up to 1.32 MBytes of additional data per frame, may also need to be transferred from the main memory.

Fig. 10 depicts the off-chip data transfers per instruction executed. At most 0.007 bytes need to be transferred from the main memory for the execution of each instruction. This proves both the effectiveness of 2D tiling in terms of data reuse, and the compute intensity of the application.

The rate is significantly lower for the Core 2 Quad than for the Cell BE, due to the larger combined outer-level cache of the former (6MB L2 in Core 2 Quad vs 2MB LSs and 512 KB L2 in Cell).

For the Cell BE, Fig. 10 reveals that the amortization of the inverse mapping cost results to more transfers per instruction. This is expected, since the LSs cannot accommodate both the input and output data for each frame, including the fractional coordinates. As a consequence, the coordinates always need to be transferred from the main memory on a tile-per-tile basis. At a first glance the diagram also seems to indicate that the Cell code is less memory efficient when the low-level optimizations are applied. This is however not true; the observed difference can be attributed to the significant reduction in the number of instructions that comes with SIMDization and loop unrolling.

It should be noted that the discussion in this subsection refers to average memory bandwidth requirements. However memory transfer requests often tend to be bursty, causing stalls, should the memory subsystem prove incapable of efficiently serving the bursts. This issue is discussed in the following subsection.

C. Analysis of stalling time

The number of stall cycles is a metric that quantifies execution delays due to either resource shortage or architectural bottlenecks. Fig. 11 illustrates the total number of stall cycles on the Core 2 Quad and the Cell BE under different degrees of optimization. Apart from the totals, it also reports the stall cycles due to two major delay factors, namely the interaction with the memory subsystem and the mispredicted branches. It has to be mentioned that other events, such as delayed bypasses, DTLB misses or blocked loads, which also contribute to the total number of stalls, are not reported individually in the charts, since their individual contribution is less profound.

It can be easily observed that the Cell BE is, due to its purposely simple architectural design, less forgiving to suboptimal software. The number of stall cycles is significantly higher for the tiled version of the code, on the Cell BE than on the Core 2 Quad. However, the low-level optimizations result to the elimination of most stalls on the Cell BE. At the same time, they reduce stall cycles by approximately 50% on the Core 2 Quad.

The sophisticated branch predictor of the Intel processor

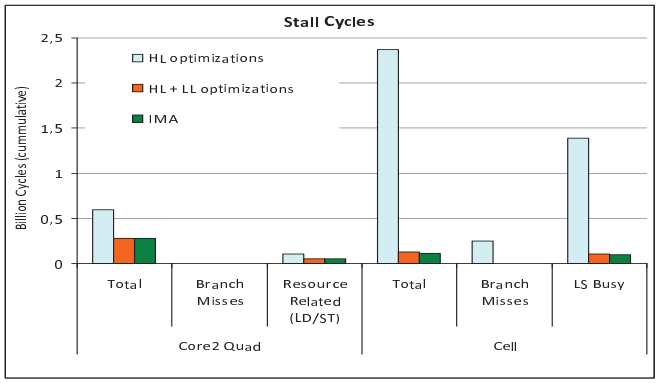


Fig. 11. Stall cycles (total, branch related and memory related) for the Core 2 Quad and the Cell BE. The number of stalls is cumulative (from all active execution contexts) and has been observed when executing with the maximum concurrency supported by each platform (4 threads on the Core 2 Quad, 8 SPEs on the Cell BE). The results are indistinguishable when less execution contexts are used.

manages to practically eliminate all stalls related to branch mispredictions. On the contrary, the Cell BE does not have a branch prediction unit. It predicts all branches as not taken - unless explicitly hinted otherwise by software - and charges a penalty of 20 cycles for mispredicted branches, whereas the typical instruction latency is 2 to 7 cycles. It should be noted that this penalty is always paid in the case of backward loop branches. The low-level optimizations, and more specifically loop unrolling and SIMDization, reduce the number of loop iterations, and as a consequence the number of backward branches. The branch-related stalls are further reduced (and almost eliminated) by software hinting. A significant contributor of stall cycles is the interaction between the processor and the memory hierarchy. Bursty memory access patterns may introduce stalls due to conflicts for cache ports, or shortage of slots in the load and store queues of the processor. The problem is worse on the Cell BE, where the Local Store is single-ported. SPE initiated requests may contend with each other, as well as with DMA transfers. Once again, low-level optimizations result to a significant reduction of memory related stalls. Loop unrolling allows the production of more efficient instruction schedules. Manual instruction scheduling, especially in the case of Cell BE, also reduces conflicts in cases where the compiler proves overly conservative.

DMA transfers between LSs and the main memory may also introduce delays due to either contention for EIB channels, or due to the DMA transfer latency itself. Double buffering proves very effective in overlapping these delays with computation. The CPC tool reported a limited cumulative number of stalls attributed to DMA transfers, which are approximately 600 cycles in the worst case.

D. Development Cost

Development cost is increasingly recognized as a significant component that needs to be considered when adopting a new platform for application development. We measure programming effort as one aspect in the comparison of the programming models of the three platforms. Because it is difficult to get accurate development-time statistics for coding applications and also to measure the quality of code, we use *Lines-of-Code* (LOC) as our primary metric to quantify programming effort.

The initial single-threaded C version was approximately 800 lines of source code. The fully optimized Cell version required an extra 1500 LOC, while the fully optimized x86 code required only 500 lines of extra code. The FPGA sDFG was written using approximately 800 lines of assembly-like instructions. Moreover, the FPGA implementation required multiple time-consuming synthesis, place & route iterations which should also be counted in the total development effort.

Based on these findings, the Core 2 Quad architecture seems to have better programmability in general, whereas the Cell processor has a slight advantage over the FPGA. However, based on the total measured development time, we think that designing and implementation of reconfigurable systems using high level languages (an area of intense research the last few years) will be very competitive in terms of development effort compared to multicore and manycore systems.

VI. RELATED WORK

The attention of both high-performance and general-purpose computing has lately turned to multicore systems, due to the diminishing performance returns of increasing processor clock frequency, and the associated power consumption and heat dissipation problems. At the same time, hardware accelerators - such as FPGAs, GPUs, or non-conventional multicore architectures such as the Cell BE - are often used to improve performance of computationally demanding algorithms or applications with execution time constraints.

Baker *et al.* describe the implementation of a matched filter on an FPGA, the Cell BE and a GPU [4]. Similarly, Thomas *et al.* implement a random number generator on a CPU, GPU, FPGA and a massively parallel processor (MPP) [26]. In [28] the authors describe the implementation of the map-reduce programming model on FPGAs and GPUs and in [17] they do the same for the Cell BE. The map-reduce infrastructure is consecutively used for the implementation of simple applications.

All the aforementioned papers focus on macroscopic metrics, such as speedup over a CPU and price / performance or power / performance ratios. In this paper we analyze the algorithm / hardware interaction using both macroscopic and low-level performance metrics across different platforms. We also identify and quantify the effects of optimizations both within and across architectures. Our work is targeted towards whole system performance instead of focusing on a specific system block. The main drawback of less conventional architectures - such as the Cell BE, GPUs and FPGAs - compared with general-purpose CPUs, is that algorithm implementation is a significantly more labor-intensive task. Previous work has focused on programming models and support to facilitate implementations. CellSs [21] and [24] introduce programming models, compiler and runtime support for task and data

management on the Cell BE. Sequoia [9] and RapidMind [20] do the same for systems with explicitly managed memory hierarchies, such as the Cell BE and GPUs.

VII. CONCLUSIONS

Modern conventional multicores and hardware accelerators - such as the Cell BE, or FPGAs - offer unprecedented computational power that allows the efficient execution of applications with high computational requirements and stringent time constraints, which previously required high-performance computing substrates or custom hardware (ASIC) implementations. In this paper we presented the implementation of a real-time image warping algorithm - with many real-world applications - on three architectures: i) a conventional, homogeneous, Intel-based multicore, ii) an heterogeneous multicore with SIMD accelerator cores (Cell BE), and iii) an FPGA. We analyzed and characterized the performance of the algorithm on all underlying architectures using both macroscopic and low-level performance metrics. We also applied a series of high- and low-level optimizations and indentified their effect on both performance and the interaction with the hardware.

We find that conventional multicores are not capable of supporting real-time video distortion correction, at least not with the currently commercially available core-counts per package. More exotic architectures, such as the Cell BE and FPGAs offer the necessary computational power, at the cost of significantly higher development effort. This additional effort can, however be partially alleviated by advanced tools, development models and support environments that allow the developer to focus on accurately expressing the algorithm, rather than on low-level optimizations.

REFERENCES

[1] *Cell B.E. Performance Counters Tools.* [Online]. Available: http://www.ibm.com/developerworks/edu/pa-dw-pasdk3tool.html

[2] *Virtex-4 Handbook*, August 2004. [Online]. Available: www.xilinx.com

[3] S. Amarasinghe and B. Thies, "Architectures, Languages and Compilers for the Streaming Domain," *Tutorial at the 12th Annual International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2003.

[4] Z. K. Baker, M. B. Gokhale, and J. L. Tripp, "Matched filter computation on fpga, cell and gpu," in *FCCM '07: Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Washington, DC, USA, 2007, pp. 207–218.

[5] N. Bellas, S. M. Chai, M. Dwyer, D. Linzmeier, and A. L. Lagunas, "Proteus: An architectural synthesis tool based on the streaming programming model," in *FPL '09: Proceedings of the 19th Conference on Field Programmable Logic and Applications*, Prague, The Czech Republic, 2009.

[6] N. Bellas, S. M. Chai, M. Dwyer, and D. Linzmeier, "Real-time fisheye lens distortion correction using automatically generated streaming accelerators," *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, pp. 149–156, 2009.

[7] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Computing Surveys*, vol. 34, no. 2, 2002.

[8] K. Daloukas, C. D. Antonopoulos, and N. Bellas, "Implementation of a wide-angle lens distortion correction algorithm on the cell broadband engine," in *ICS '09: Proceedings of the 23rd International Conference on Supercomputing*, New York, NY, USA, 2009, pp. 4–13.

[9] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: programming the memory hierarchy," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, New York, NY, USA, 2006, p. 83.

[10] M. S. Friedrichs, P. Eastman, V. Vaidyanathan, M. Houston, S. LeGrand, A. L. Beberg, D. L. Ensign, C. M. Bruns, and V. S. Pande, "Accelerating molecular dynamic simulation on graphics processing units," *Journal of Computational Chemistry*, vol. 30, no. 6, pp. 864–872, 2009.

[11] M. B. Gokhale, J. M. Stone, J. Arnold, and M. Kalinowski, "Stream-oriented fpga computing in the streams-c high level language," in *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, Washington, DC, USA, 2000, pp. 49–56.

[12] Intel Corporation, *Intel Thread Profiler*, Document Number 300638-001.

[13] Intel Corporation, *Intel VTune Performance Analyzer*, Document Number 310866-001.

[14] N. D. Jankovic and M. D. Naish, "Developing a modular active spherical vision system," in *Proceedings of the 2005 IEEE International Conference on Robotics and Automation, Barcelona, Spain*, April 2005.

[15] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM Journal of Research and Development*, vol. 49, no. 4/5, pp. 589–604, 2005.

[16] R. Keyes, "Cubic convolution interpolation for digital image processing," in *IEEE Transactions on Acoustics, Speech ,and Signal Processing*, vol. ASSP-29, no. 6, December 1981.

[17] M. Kruijf and K. Sankaralingam, "Mapreduce for the cell b.e. architecture," University of Wisconsin, Computer Siences Technical Report CS-TR-2007-1625, October 2007.

[18] D. P. Kuban, H. L. Martin, S. D. Zimmermann, and N. Busko, "Omniview motionless camera surveillance system," US Patent 5 359 363, Feb., 1993.

[19] D. Lau, O. Pritchard, and P. Molson, "Automated generation of hardware accelerators with direct memory access from ansi/iso standard c functions," in *FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Washington, DC, USA, 2006, pp. 45–56.

[20] M. Monteney, "RapidMind multi-core development platform," White Paper, RapidMind Inc., February 2008.

[21] J. P. Perez, P. Bellens, R. M. Badia, and J. Labarta, "CellSs: making it easier to program the cell broadband engine processor," *IBM Journal of Research and Development*, vol. 51, no. 5, pp. 593–604, 2007.

[22] F. Petrini, G. Fossum, J. Fernandez, A. L. Varbanescu, M. Kistler, and M. Perrone, "Multicore surprises: Lessons learned from optimizing sweep3d on the cell broadband engine," in *Proceedings of the 2007 International Parallel and Distributed Processing Symposium (IPDPS 2007)*, March 2007.

[23] F. Plavec, Z. Vranesic, and S. Brown, "Enhancements to FPGA design methodology using streaming," in *FPL '09: Proceedings of the 19th Conference on Field Programmable Logic and Applications*, Prague, The Czech Republic, 2009.

[24] S. Schneider, J.-S. Yeom, B. Rose, J. C. Linford, A. Sandu, and D. S. Nikolopoulos, "A comparison of programming models for multiprocessors with explicitly managed memory hierarchies," in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Raleigh, NC, February 2009.

[25] E. Schwalbe, "Geometric modeling and calibration of fisheye lens camera systems," in *Proceedings of the ISPRS Working Group, Panoramic Photogrammetry Workshop, Berlin, Germany*, vol. 34-5/W8, February 2005.

[26] D. B. Thomas, L. Howes, and W. Luk, "A comparison of cpus, gpus, fpgas, and massively parallel processor arrays for random number generation," in *FPGA '09: Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, 2009, pp. 63–72.

[27] T. Yamamoto and M. Doi, "Design and implementation of panoramic movie system by using commodity 3d graphics hardware," *Computer Graphics International Conference*, pp. 14–19, 2003.

[28] J. H. C. Yeung, C. C. Tsang, K. H. Tsoi, B. S. H. Kwan, C. C. C. Cheung, A. P. C. Chan, and P. H. W. Leong, "Map-reduce as a programming model for custom computing machines," in *FCCM '08: Proceedings of the 2008 16th International Symposium on Field-Programmable Custom Computing Machines*, Washington, DC, USA, 2008, pp. 149–159.

[29] S. Zimmermann and D. Kuban, "A video pan/tilt/magnify/rotate system with no moving parts," in *IEEE Digital Avionics Systems Conference*, October 1992, pp. 523–531.