Implementation of a Wide-Angle Lens Distortion Correction Algorithm on the Cell Broadband Engine

Konstantis Daloukas
Department of Computer and Communications Engineering
University of Thessaly
Volos, Greece
kodalouk@inf.uth.gr

Christos D. Antonopoulos
Department of Computer and Communications Engineering
University of Thessaly
Volos, Greece
cda@inf.uth.gr

Nikolaos Bellas
Department of Computer and Communications Engineering
University of Thessaly
Volos, Greece
nbellas@inf.uth.gr

ABSTRACT

Wide-angle lenses are often used in scientific or virtual reality applications to enlarge the field of view of a conventional camera. Wide-angle lens distortion correction is an image warping application which transforms the distorted images back to the natural-looking, central perspective space. This application is characterized by a non-linear streaming memory access pattern that makes main memory bandwidth a key performance limiter.

This paper presents the implementation, optimization and evaluation of a wide-angle lens distortion correction algorithm on the Cell Broadband Engine. Cell is a heterogeneous multi-core processor that has been architected to accelerate applications with large degree of thread- and data-level parallelism. We alleviate the 'memory wall' inefficiencies by applying source level optimizations such as tiling to better utilize the on-chip memory of the SPE, and maximize data reuse within a frame of pixel data. Using these transformations on the Cell processor, we are able to achieve a 7.27x speedup compared to a Core2 Duo processor, and enable potential applications such as real-time correction for video teleconferencing using cameras with wide-angle lenses. To the best of our knowledge, this is the first paper to describe the mapping and optimization of an image warping application to the Cell BE architecture.

Categories and Subject Descriptors

I.4.9 [**Image Processing and Computer Vision**]: Applications; C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Heterogeneous (hybrid) systems*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

General Terms

Algorithms, Design, Measurement, Performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICS'09, June 8–12, 2009, Yorktown Heights, New York, USA.
Copyright 2009 ACM 978-1-60558-498-0/09/06 ...$5.00.

Keywords

Image Warping, Real-Time, Parallel Programming, Stencil Computation, Cell B.E., Heterogeneous Multi-Core Processors

1. INTRODUCTION

Wide-angle lenses allow imaging a large sector of the surrounding space with a single photo. While ordinary rectilinear lenses map incoming light rays to a planar photosensitive surface, wide-angle lenses map them to a spherical surface, which is capable for a much wider field of view (FoV). It is possible, and in fact very common, for wide-angle lenses to encompass a FoV of 180°. Such hemispherical images have been traditionally used for specialized applications such as surveillance [16], robot navigation [6], content creation for immersive environments and virtual reality [19], photography [21], astronomy, etc. In addition, wide-angle imaging is moving towards more mainstream applications such as consumer digital imaging and video capture, or even real-time video conferencing. By capturing a larger section of the surrounding space, a wide-angle lens camera affords a wider horizontal and vertical viewing angle, provided that the distorted images at the wide-angle space can be corrected and transformed into the central perspective space in real time before being viewed by the end user (Figure 1). Real-time distortion correction for megapixel input frame resolution is beyond the reach of today's mainstream general purpose processors, as we show in section 3.

In this paper, we first explain the algorithmic aspects of wide-angle lens distortion correction in section 2. In section 3 we describe the mapping and optimization of a wide-angle lens distortion correction algorithm on the Cell Broadband Engine (CBE) heterogeneous multicore processor.

The CBE offers a rich repertoire of mechanisms to exploit thread and data level parallelism owing to eight Synergistic Processing Elements (SPEs) and a 2-way SMT PowerPC core (PPE) [7]. The processor is clocked at 3.2 GHz. Each SPE offers a 128-bit wide SIMD computational engine and has access to a private, 256 KB software-controlled local store (LS), which is shared by both program code and data. SPEs can access the main memory only through asynchronous DMA requests. Up to two 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 floating point operations.

This paper explores how the inherent parallelism of the wide-angle lens distortion correction algorithm is exploited

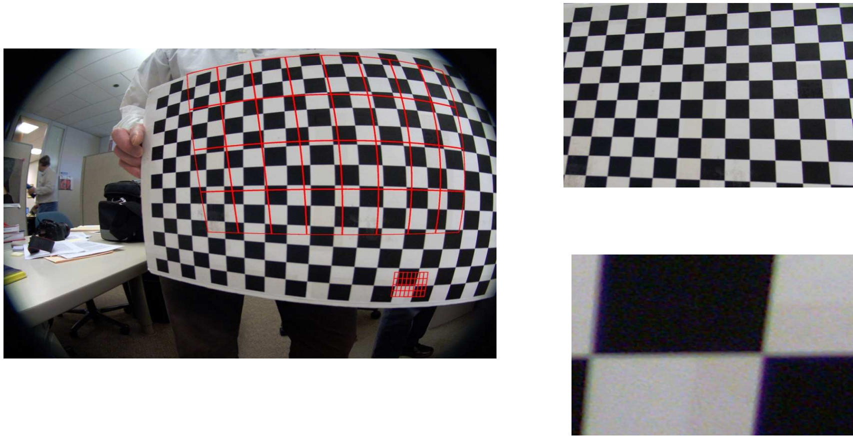


Figure 1: The wide-angle lens distortion correction algorithm for two cases for field of view FoV=60° and FoV= 8° . The output images are VGA (640x480). The lower FoV results into a zoomed output image.

on the CBE architecture to achieve real time functionality at 30 frames/sec for megapixel input frames. We present a quantitative analysis on the impact of each optimization on application performance, using a real Playstation3 (PS3) based on the Cell processor. Although the image warping 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. We make use of the high local memory bandwidth available in SPEs by organizing the computations around tiled datasets. The use of the local store (LS) memory provides plenty of bandwidth to feed the SIMD engine of each SPE, yet the lack of availability of non vector-aligned loads and stores places a significant burden on the programmer, who has to tune the code to match the size and alignment of the SPE architecture. This can limit Cell performance for a series of multimedia applications that are characterized by random, non-aligned memory accesses.

Another interesting finding is that the SPE code required a significant degree of manual instruction scheduling to reduce pipeline stalls due to data dependencies and slot misalignments. This illustrates the need for more mature SPE compiler technology capable of producing optimized code. The aforementioned optimizations are applicable to many stencil computation codes.

Mapping an algorithm with inherent multilevel parallelism on a complex, heterogeneous parallel architecture, like the CBE, requires the developer to make numerous design decisions. Section 4 presents the most interesting " what-if " scenarios and how they affect performance.

The contribution of this paper lies on introducing a complex image processing application, explaining the source level optimizations on the original code to exploit the heterogeneous multicore architecture of CBE and evaluating the performance under numerous implementation scenarios. The detailed empirical optimizations we outline are highly unlikely to be made automatically by an optimizing compiler, especially since they require the application of high level compiler transformations to the original, sequential code. We also identify some counter-intuitive optimizations that minimize the effects of unaligned memory accesses in the local store of SPEs.

2. WIDE-ANGLE LENS DISTORTION CORRECTION ALGORITHM

The stereoscopic geometry of wide-angle photography does not comply with the conventional central perspective projection shown in Figure 2(a)[15]. 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° 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[1] is based on the principle that the incidence angle is proportional to the distance between the image point and the central point i.e. $\frac{d_1}{d_2} = \frac{a_1}{a_2}$ (Figure 2(b)). 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:

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

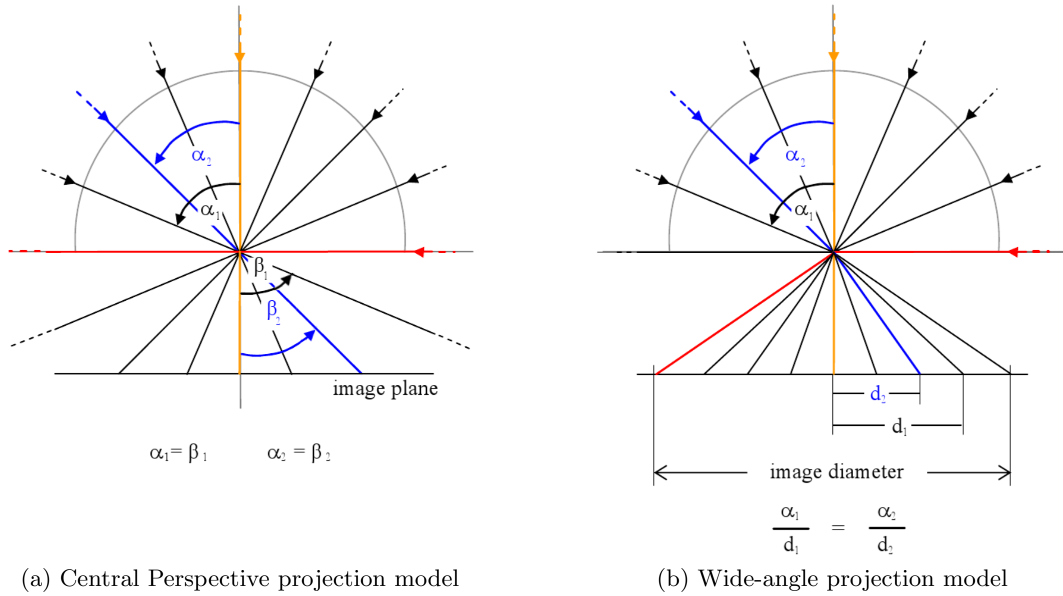
After some algebraic transformations [15], the equations that describe the projection on the image plane when using wide-angle lens are:

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

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

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

[1]Also known as fisheye projection in the literature



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

Figure 2: Projection models used in camera systems. Each ray passes through the principle point. The optical axis is depicted in orange.

the image radius. Equations (1) provide a method to convert the 3D central perspective space coordinates (X_c, Y_c, Z_c) of an object back to the distorted 2D wide-angle space (inverse mapping). They can be broken into elementary mathematical functions and the lens distortion parameters can be folded into the equations:

$$d = \sqrt{X_c^2 + Y_c^2}, D_u = \frac{d}{Z_c}, R_u = \arctan D_u \tag{2a}$$

$$P = k_1 \cdot R_u^4 + k_2 \cdot R_u^3 + k_3 \cdot R_u^2 + k_4 \cdot R_u + k_5 \tag{2b}$$

$$x = \frac{P}{d} \cdot X_c + x_h, y = \frac{P}{d} \cdot Y_c + y_h, \tag{2c}$$

where k_i are lens parameters.

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. We use bicubic interpolation [9] to approximate the pixel value at that fractional point, which equals the pixel value at the initial (*i, j*) location in the central perspective space.

Bicubic interpolation is a robust, yet computationally expensive technique. It uses cubic sampled functions to approximate intermediate points of a continuous event, given the interpolation nodes C_i where the interpolation function *g()* is known to equal the (unknown) interpolated function *f()*. The following equation approximates the value of a function *f()* at point x, based on known sampled point values $C_i = f(x_i)$:

$$g(x) = C_1 \cdot U_1(s) + C_2 \cdot U_2(s) + C_3 \cdot U_3(s) + C_4 \cdot U_4(s)$$

$$U_1 = \frac{-s^3 + 2s^2 - s}{2}, \quad U_2 = \frac{3s^3 - 5s^2 + 2}{2}$$

$$U_3 = \frac{-3s^3 + 4s^2 + s}{2}, \quad U_4 = \frac{s^3 - s^2}{2}$$

where the point x is such that $x_1 \leq x_2 \leq x \leq x_3 \leq x_4$ and $s = x - x_2$.

Although other techniques, such as nearest neighbor or bilinear are simpler, the high *Signal to Noise (SNR)* requirements of the correction algorithm, especially for low *FoV* angles, makes this the method of choice. More computationally expensive methods like the *Elliptical Weighted Average (EWA)* [5], require computing complex power values, which can tax the ability of the FP unit to process frames in real time.

The inverse mapping and 2D bicubic interpolation flows are shown in Figure 4. Note that the algorithm starts by enumerating the (*i, j*) pixel coordinates at the perspective space and then maps these coordinates back to the wide-angle space. The approximation (using bicubic interpolation) of the pixel value at these fractional points is equal to the value of the corresponding pixel at the perspective space. The two-dimensional bicubic interpolation method consists of an one-dimensional interpolation in each dimension. The method requires the use of the 16 pixel values of a *4x4* window around the interpolated point.

In order to eliminate high frequency artifact noise on the image, we apply a 5-tap vertical and a 5-tap horizontal low pass filter on the corrected image to downsample the image to the resolution requirements of the target application, in our case *640x480 VGA*.

Figure 3 outlines the high-level pseudocode of the distortion correction algorithm.

3. MAPPING AND OPTIMIZATION

An important observation from the algorithmic analysis of section 2 is that the fractional pixel coordinates follow a complicated non-linear pattern (Figure 4). The exact trace shape depends on a variety of factors, including the FoV, the location of the pixel, and parameters modeling lens distortion. Although the trace is not data dependent, and thus, can be theoretically pre-computed, the complex memory ac-

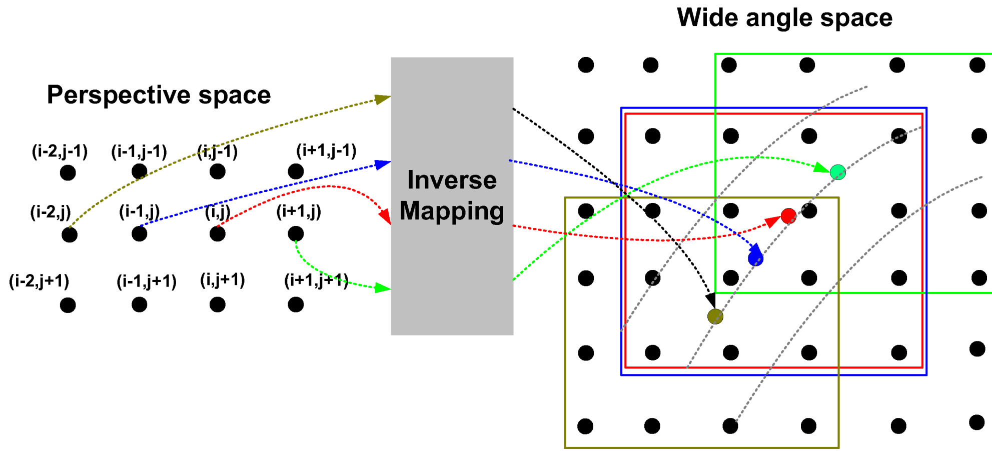


Figure 4: Inverse mapping is used to convert the coordinates from the perspective space back to the wide-angle space. A 4x4 pixel neighborhood around the fractional points on the distorted space is used to perform bicubic interpolation and compute the pixel values at the fractional points.

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**

Figure 3: Fisheye lens distortion correction algorithm (original version).

cess pattern deems aggressive DMA prefetching impractical.

The algorithm has a large degree of data reuse but not necessarily across a row or column of the frame. Reuse is maximized 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 tiling computations to exploit reuse at the block level, we also facilitate data distribution to *SPE Local Stores*.

Figure 5 shows a block of output pixel data from the frame in Figure 1, and the corresponding curved bounding block of pixels at the distorted space (shown in red), needed to interpolate the output pixels. Adjacent rectangular boxes in the fisheye space will partially overlap, due to the curvature of the distorted boxes. As a result, pixels close to the edges of the bounding box are being fetched from the memory more than once.

The input image in our experiments (Figure 1) is full RGB with size 2592x1944, but the inverse mapping, bicubic interpolation and low-pass filter are applied on a 4xVGA image (1280x960) since we always work within a region of interest (ROI). The output image is then downsampled to VGA resolution.

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

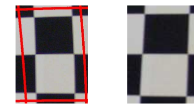


Figure 5: Each SPE works on a tile of pixel data and produces a rectangular pixel block. The input tiles (left) contain extra pixels beyond the red boundary, so that their shape is always rectangular.

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.

In order to evaluate whether achieving real-time wide-angle lens distortion correction on a conventional, general-purpose processor is a realistic undertaking, we executed a parallel, optimized version of the code on a system based on an Intel T7500 Core2 Duo processor, clocked at 2.2 GHz. The system is equipped with 2 GB of RAM and runs Linux (OpenSuse 10.3 with a 2.6.22 kernel). We used both the *icc* from the Intel compiler suite and *gcc*, using the compiler flags that resulted to the lowest execution times. The performance of executables produced by *icc* proved slightly higher, so we report only these results.

The code on the Intel-based platform has been parallelized using OpenMP. Each tile of the output image is computed independently, potentially by a different execution context of the processor. In addition to tiling the input and output frames, we have performed all applicable optimizations described later in this section, namely manual vectorization using the SSE ISA extensions and loop unrolling, as well as careful manipulation of unaligned loads to vector registers.

The application processes 2.38 frames/sec (fps) using one thread, raising to 4.17 fps when two threads are used, far from the minimum requirement of 25-30 fps for real-time processing of video streams. These results motivate the use of a high-performance, non-conventional processor such as the CBE.

The rest of the chapter describes the porting of the ap-

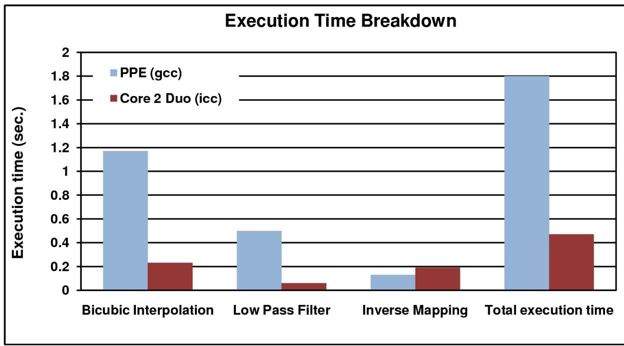


Figure 6: Program execution time for the PPE (with gcc) and Core2 Duo (with icc).

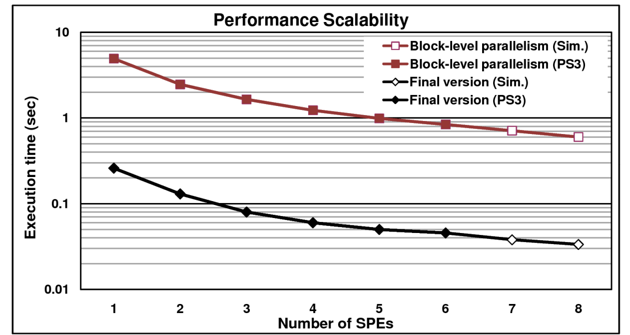


Figure 7: Execution time when (a) we exploit task level parallelism w/o double buffering and (b) for the final, optimized version of the code described later in section 4.3. Note that the graph is in logarithmic scale. The data points for seven and eight SPEs are obtained using the Cell simulator. The performance reported by the simulator for 1-6 SPEs coincides with measurements on the real machine.

plication on the Cell processor, taking the following steps: (1) the transformation of the initial frame-based code into a block-based version, (2) the porting of the new code version on the PPE only, (3) the offloading of the inverse mapping, bicubic interpolation and low pass filter functions to the SPEs, and (4) the optimization of the SPE code by using vectorization, optimizing LS unaligned accesses, manual instruction scheduling to reduce stalls, and overlapping of DMA accesses with SPU execution.

For our experimental evaluation, we used a PS3 platform in which only six out of the eight SPEs of the Cell processor are available. The systems runs Yellow Dog Linux 6 with a 2.6.23 kernel. All software runs on top of a hypervisor, which does not allow user access to the hardware performance counters of the processor. Therefore, to obtain more detailed results, and evaluate performance on eight SPEs, we used the cycle-accurate full-system Cell simulator under Fedora Linux 7. The latest version (3.1) of the simulator is cycle accurate for almost all elements of the system. We compiled the source code using both *gcc* (version 4.2.1) and *xlc* compiler (version 9.0), using in each case the compiler flags which resulted to the fastest executables. The performance of the executables produced by *gcc* was higher, so we only report results using *gcc*.

3.1 Porting to the PPE

The first step in mapping the code to the Cell processor is to apply source level transformations to re-arrange the processing in tiles. We profiled the optimized code to identify the most computationally demanding functions. The code was profiled on a single-threaded execution on both the Core2 Duo system and on the PPE Cell processor (without *Altivec* extensions).

The execution time of the code is mainly spent in three functions: *inverse_mapping*, *bicubic_interpolation*, and *low_pass_filter*. Figure 6 depicts the breakdown of the execution time on the two processors.

The results show that bicubic interpolation is the most time-consuming kernel on both processors accounting for more than 60% of the execution time in all cases. Inverse mapping is very efficient, owing to the fast FP units of the Core2 Duo and the PPE. Core2 Duo outperforms the Power Processing Element (PPE), thus the PPE is also incapable of handling real-time distortion correction. In the following subsections, we discuss the sequence of coarse and fine-grain optimizations we applied on the code to achieve real-time functionality on the CBE.

Note also that the inverse mapping kernel does not need to be invoked for every frame, and it is only called if the ROI or the FoV parameters change. For a realistic usage scenario, we do not expect that to happen very often, at least no more frequently than once every few seconds. The results depicted in Figure 6 assume that the inverse mapping is computed for every frame. Later, in section 4 we evaluate the case where we execute inverse mapping only once, store the results to the main memory, and transfer the fractional coordinates to the SPEs for each tile.

3.2 Block-level Parallelism

The three functions *inverse_mapping, bicubic_interpolation*, and *low_pass_filter* are mapped to the SPEs since they are responsible for the vast majority of execution cycles (Figure 6). The PPE runs the initialization code, and spawns six threads to be executed on each SPE.

We exploit the independence between the tiles of the output image and we assign tiles to SPEs, using a data-cyclic partitioning scheme. Each SPE fetches a new block of input data from the main memory, applies the three functions on the pixel data and writes the results back to main memory. We set the tile size to be 256 columns by 48 rows. We experiment with different tile sizes in section 4 to determine their effect on the execution time.

The DMA requests are initiated by the SPEs. The aforementioned partitioning enables the SPEs to work independently from other SPEs and from the PPE and the only explicit synchronization is at the end of each frame.

Figure 7 shows that there is significant performance gain when we increase the number of participating SPE accelerators, due to the high degree of data level parallelism at the block level. In fact, the execution time drops linearly with the number of SPEs from 4.93 secs/frame down to 0.62 secs/frame (speedup is 8x). Figure 7 also shows that the SPE architecture is not optimized to run scalar code as efficiently as the PPE architecture: a single PPE thread of execution processes one frame every 1.8 secs, whereas a single SPE thread processes the frame in 4.93 secs.

Table 1: Execution time, speedup factor and branch misprediction overhead (a) before any SPE optimization, (b) after vectorization and 4x outer loop unrolling, and (c) after 3x inner loop unrolling. In all cases, all 6 SPEs of the PS3 platform are used.

Optimization	Exec. Time (sec)	Speedup over (a)	Branch misprediction stalls (% cycles)
(a) Coarse grain parallelization	0.84	1	12.5%
(b) Vectorization and 4x unroll	0.093	9.03	3.1%
(c) 3x inner loop unroll	0.07	12	2.6%

3.3 DMA Double Buffering

In the unoptimized version of the code, SPE computation does not overlap with DMA transactions. This adds a small performance overhead of 0.005 secs/frame to the execution time, since the high available bandwidth of the EIB bus (96 B/cycle) ensures that all DMA transactions are served quickly.

Nevertheless, we accelerate the previous process by allocating two separate buffers, and overlapping computation for tile N from one buffer with incoming input transfers for tile N+1 and outgoing result transfers for tile N-1. This double buffering technique effectively hides memory access latency at the expense of increased pixel storage requirements and code complexity. The elimination of this small overhead is, however, more important in the final, optimized version of the code.

3.4 Vectorization and loop unrolling

The next step is to optimize the code executed on the SPEs. We are using a series of manual software transformations to take advantage of features such as the vector execution units and the dual issue pipeline, as well as more unconventional optimizations that would not typically be applied in a scalar processor.

Most calculations in each SPE processor 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 filtering, each enclosing a second-level nested loop with three iterations.

We utilize the vectorization capability of the SPEs by clustering four single precision FP operands in a vector register and operating on them in parallel. Under this scheme, inverse mapping produces four coordinate pairs in an iteration, the one-dimensional bicubic interpolation stencil is applied to four pixel values per iteration and so on. Since SPEs spend all of their execution time within the outer loops of these three functions, the SIMD parallelization alone can potentially speed up the program by 4x.

The implicit loop unrolling due to vectorization has the additional positive effect of reducing the backward branches of the outer loop by a factor of 4. Mispredicted branches incur a large penalty of up to 20 cycles whereas the typical instruction latency is only 2 to 7 cycles. This penalty is always paid in the case of backward loop branches, since the SPE always predicts branches as not taken, unless instructed differently by the compiler or the programmer. To further the positive effects of branch elimination and increase the potential for efficient instruction scheduling, we unroll all the inner loops (for R,G,B) three times, effectively eliminating them.

Table 1 shows that the first optimization has a cumulative speedup of 9.03 over the unoptimized SPE code, and reduces the stall overhead due to mispredicted branches from 12.5% down to 3.1% of the total cycles. The unrolling of the inner loops further reduces the execution time to 0.07 sec/frame for a total 12x speed up. Further unrolling the outer loops did not result into significant gains.

3.5 Unaligned loads

One of the challenges to SIMD vectorization is that the bicubic interpolation requires memory accesses to addresses that are not vector aligned (Figure 4). In the case of scalar loads, the SPE compiler inserts extra assist and shift instructions in order to move the scalar data to the preferred slot. The static profiling of the SPE pipeline indicated that the majority of the remaining pipeline stalls were due to the unaligned memory accesses in the formation of the 4x4 window. The four pixels of each row − for a single color channel − are loaded individually in a single precision floating point vector register. The conversion from integer to single precision floating point is needed because subsequent operations on the pixels use FP arithmetic. Since each vector register load is performed atomically for each pixel in the row, data dependencies from previous loads stall consecutive loads on the same register, making the vector register a point of contention.

We reverse the loading of pixels to the vector register by loading first the first column of pixels (on four different registers, one per row), then the second column, and so on. We still use a single vector register to store each row of pixels, but we change the sequence in which the four registers are filled (Figure 4). This modification spreads out the instructions that access the same register, allowing enough time to load a pixel into one register, before we load the next pixel to the same register. Note that this is a counter-intuitive schedule, since compiler optimization techniques tend to arrange data accesses in rows. This optimization results into a further reduction of the execution time to 0.06 secs/frame.

3.6 Vectorizing conditional statements and manual scheduling

The *bicubic_interpolation()* function includes the following conditional statement to check whether a given location (X, Y) is outside the frame boundaries. If this is the case, the corresponding pixel is set to black in the array *lpf*, which serves as an intermediate buffer and stores the results of the vertical low-pass filter:

```
if ((X < 0) || (X > (Width - 1)) ||
    (Y < 0) || (Y > (Height- 1)))
    lpf[loc][0]=lpf[loc][1]=lpf[loc][2]=0;
```

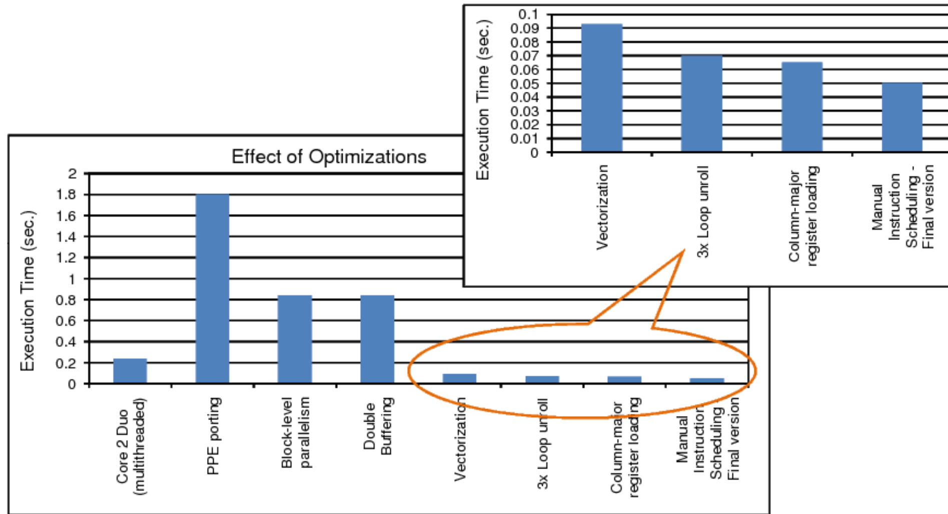


Figure 8: Wide-angle lens distortion correction performance on the Core2 Duo and the CBE under all different optimization scenarios. The final version requires only 0.05 secs to process a frame and can achieve 20 fps. All the numbers refer to real execution times on the PS3 platform.

After SIMD optimization, there are four such if-statements in the outer loop and four X and Y coordinates reside in two different vector registers. 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 instructions inside the if-statement.

Our approach is to vectorize the calculation of the conditional and move it at the beginning of the outer loop. We still have to extract the scalar conditional values, but this is done at the beginning of the outer loop, enough cycles before these values are needed.

As a last optimization step, we eliminate most of the remaining pipeline stalls by manual scheduling of instructions. We interleave unaligned loads with computational operations, thus enabling the compiler to schedule the instructions more efficiently. Manual scheduling is facilitated by the previous 3x unrolling optimization which provide a large number of instructions that can be interleaved. It should be noted that the compiler proved too conservative rescheduling independent instructions only locally.

Figure 8 illustrates the performance of the application after each one of the optimizations described in this section. Exploiting coarse-grain data-level parallelism across SPEs and fine-grain parallelism like vectorization, and elimination of backward branch instructions were the most successful optimizations. The results show that although the single- and multi-threaded execution on Core2 Duo clearly outperform the PPE by a margin of 4.29x and 7.5x respectively, the combination of the PPE (as control processor) and 6 SPEs is 4.8x faster than the multi-threaded execution on the Intel processor when the code is optimized (see bottom line in Figure 7). This observation confirms the capability of the CBE architecture to exploit multilevel parallelism, at the expense of extensive source level optimizations.

4. DESIGN TRADE-OFFS AND SENSITIVITY ANALYSIS

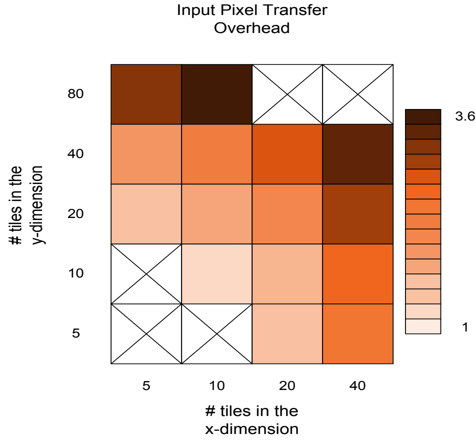
In this section we discuss various trade-offs we faced while developing the wide-angle lens distortion correction application on the CBE.

4.1 Tile size

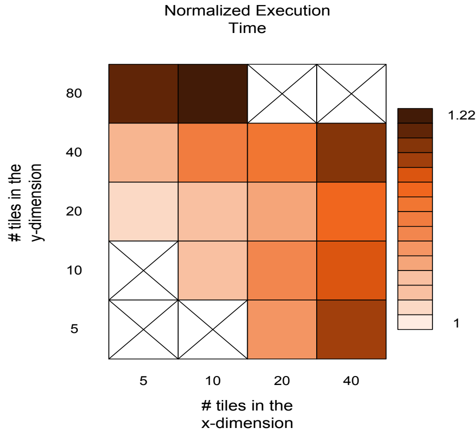
Tile size and shape is a significant parameter in explicitly blocked codes. Typically, tile size should be large enough to use all available cache (or local store in the CBE), in order to maximize data reuse and minimize communication overhead. Moreover, partitioning to fewer tiles reduces the total DMA transfer setup overhead.

In the case of wide-angle lens image correction however, additional parameters, besides the local store size and the footprint of other data, should be taken into account. The area of the input frame that needs to be transferred is typically curved, as depicted in Figure 5. Since the manipulation of non-rectangular memory areas is not trivial, we fetch the enclosing rectangle of the curved area instead. Additional limitations are imposed on the size and shape of the rectangle i.e. its width should be a multiple of *4* in order to facilitate vectorization. Moreover, each line should start from a *16B* aligned address, the lowest permissible alignment to achieve DMA transfers of more than *16B* data chunks. Due to the curvature of the input area, a very small tile results to a higher overhead of pixels outside the input area of interest, that should nevertheless be transferred since they reside in the enclosing rectangle. The effect of tiles being too stretched to any dimension is similar.

The left diagram of Figure 9 depicts the overhead (pixels transferred divided by total frame pixels) of input pixels transfered for the correction of a single frame, for different legal output tile sizes and shapes, according to the aforementioned limitations. The right diagram depicts the corresponding normalized execution times on the 6 SPEs of the PS3. The two diagrams reveal a close relation among data transfers and execution time for different tile sizes. Moreover, it is clear that too small tiles or stretched tiles (upper left and lower right corners) are not preferable. The percentage of excessive pixel transfers ranges from 9% to 331%. The potential slowdown, with respect to the execution time



(a)



(b)

Figure 9: (a) Input pixels transferred to the SPE local stores divided by the number of pixels in the frame, and (b) execution time normalized with respect to the optimal execution time, when output frame is partitioned into 5x20 rectangular tiles. The white boxes with the X mark, do not correspond to a valid tile partition.

using the optimal block size, is up to 22%. Driven by these results, we used a decomposition of the output frame to 5 tiles in the x-dimension by 20 tiles in the y-dimension.

4.2 Amortization of inverse mapping cost

Inverse mapping, i.e. the procedure of calculating for each pixel in the output frame the coordinates of the corresponding fractional pixel in the input frame, is one of the three main contributors to the total execution time of the application (Figure 6). As aforementioned, the correspondence of input versus output pixel coordinates depends solely on the region of interest (ROI) and the field of view (FoV). 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

1: {**Input**: The frames (in the wide-angle space) to be corrected}
2: {**Output**: The corrected frames (in the perspective space)}
3: Partition the output frame to blocks
4: **for all** frames **do**
5: **if** FoV has changed **then**
6: **for all** pixels in the output frame **do**
7: Compute the corresponding fractional position in the input frame (`InverseMapping()`)
8: **end for**
9: Calculate the area of the input frame required for the calculation of each output frame block
10: **end if**
11: {Output blocks are statically partitioned to SPEs and processed concurrently on different SPEs}
12: **for all** output blocks i **do**
13: Fetch (async. DMA) input data for output block $i+1$
14: Fetch (async. DMA) fractional coordinates in the wide-angle space for each pixel of the output block $i+1$
15: Store (async. DMA) output block $i-1$
16: **for all** pixels in output block i **do**
17: Interpolate the pixel value at that fractional position (`BicubicInterpolation(), SIMDized`)
18: **end for**
19: Apply a 2-D low-pass filter to resize the output block (`LPF(), SIMDized`)
20: **end for**
21: **end for**

Figure 10: Fisheye lens distortion correction algorithm (final, optimized version).

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. A data structure of that size can not be accommodated in the local store of SPEs. We evaluated the option of storing the inverse mappings at main memory and transferring the appropriate block of fractional coordinates to each local store, according to the output image tile processed by the corresponding SPE.

The computation vs. communication trade-off is of particular interest, given the vast processing power of the CBE. We found that the increased communication requirements resulted, even after double buffering, to a 4x increase of the cycles the application spends waiting for the completion of DMA requests. Nevertheless, the computation time, evaluated over a sequence of 10 frames, decreased to an average of 0.045 sec/frame when all 6 SPEs of the PS3 are used, yielding a rate of more than 22 fps. The simulator reported an average of 0.033 sec per frame, or 30 fps, when 8 SPEs are available. This corresponds to a speedup of 7.27x over the multithreaded execution on Core2 Duo. Figure 10 outlines the resulting pseudocode.

Another parameter in the communication vs. computation trade-off is whether $U_i(s)$, $V_j(t)$ interpolation coefficients should be pre-computed and stored in the main memory instead of the s and t coordinates. These coefficients depend only on the values of the s and t coordinates, and can therefore be computed once in the PPE and transferred to the SPEs for the bicubic interpolation. The disadvantage of this approach is that eight such floating point coefficients for each pixel (compared to only two coordinates) should be transferred and stored to the LS. This would necessitate the use of a smaller tile size, with detrimental effects on performance as shown in section 4.1.

4.3 Input data preprocessing at the PPE

We discussed earlier that misaligned memory accesses, typical in stencil computations where the stencil sweeps over input data, is a major performance limiter in the SPEs. Each input frame is organized as a 2D array of pixels, with three bytes per pixel (RGB color intensities). The code that loads pixel values to SPE registers has to: (a) demultiplex the bytes corresponding to different colors, since each color is processed separately, and thus is loaded to a different 128-bit register, (b) upgrade 8-bit integer values to 32-bit floats, since computation is performed with single-precision floating point arithmetic, and (c) pack in the same register the data corresponding to a specific color of four consecutive pixels.

We experimented with preprocessing input data at the PPE, in order to minimize data manipulation at the SPEs. Input data are converted to floats in main memory and different colors of each frame are demultiplexed. Register loads are now aligned with a probability of 25%. In order to double the number of aligned loads, we created a second copy of the input tiles in main memory, padded by two floats. As a result, loads are aligned with a probability of 50%. However, the amount of input tile data that have to be transferred from the main memory to each SPE, and the footprint of input tiles on each local store have both been multiplied by a factor of 8.

The new version of the code executed 30% slower. We used the CBE simulator to identify the cause, which proved to be the inability to overlap communication with computation. Moreover, the preprocessing overhead made the PPE code the main bottleneck of the application, further limiting overall performance.

5. RELATED WORK

The CBE processor was originally targeted to the gaming and multimedia markets. It soon, however, attracted the interest of the scientific community, due to its vast computing power and its very appealing price / performance and power / performance ratios.

It has been successfully used in the context of medical imaging [10], computational biology [13], particle transport simulations [12], data mining [1, 3] and on-line network intrusion detection systems [14]. The exploitation of the computing capabilities made available by the CBE facilitated dealing with challenging problems that previously required expensive, large-scale computing systems, or even specialized hardware support. CBE is also a popular target for media processing applications, such as high definition video decoding [2], and speech recognition [11]. Media applications are usually computationally demanding, and are characterized by a low computation to communication ratio and often set real-time constraints. To the best of our knowledge, this is the first study of the feasibility of real-time image warping on the CBE.

In [8] and [17] the authors evaluate the performance of stencil computation kernels on Cell. They observe that the rigid requirements of data alignment imposed by the processor can be a significant, performance-limiting bottleneck for such codes. They suggest explicit blocking and time skewing [18] in order to improve locality and increase the computation versus communication ratio.

Eichenberger et al. [4] introduce compiler techniques targeted towards the automatic generation of highly efficient code for the Cell B.E. These techniques include compiler assisted memory alignment and branch prediction, SIMD and thread-level parallelization, and compiler-controlled software caching. Similarly, Zhao and Kennedy [20] introduce a research compiler for Cell, which offers automatic, dependence-driven loop-parallelization and SIMDization. Our work reveals some weaknesses of current, commercially available compilers for Cell, and outlines the importance of advanced compiler support in order to facilitate the programmability of complex, heterogeneous multi-core architectures like the CBE.

6. CONCLUSIONS

The unprecedented processing power of the Cell Broadband Engine heterogeneous multi-core processor enabled software solutions to computationally demanding algorithms with real-time constraints, previously requiring specialized hardware support. In this paper we focused on the implementation of a real-time, wide-angle lens distortion correction algorithm, an image-warping technique with applications to different scientific domains. This application is characterized by a static, yet irregular memory access pattern that makes data prefetching a challenging undertaking.

We outlined and evaluated experimentally the step-by-step algorithmic and architecture-driven optimizations that were required to achieve a 30 fps real-time performance for sequences of full RGB, 2592x1944 input and 640x480 output frames. We explored the performance implications of several coarse and fine-grain optimizations. Starting from a performance of 4.17 fps on a Core2 Duo processor, we were able to map and gradually optimize the code on the CBE, converging to the final implementation running at 30 fps. In the course of this experimental evaluation, we determined that the most profitable optimization steps were task offloading, vectorization and branch elimination.

Moreover, we explored a series of interesting " what-if " scenarios to assess the impact of alternative software transformations on the application performance. It turned out that the optimal performance is reached when the PPE executes the inverse mapping (using *Altivec*) and then distributes the values of the fractional coordinates to the SPEs. This approach attains the minimum execution time at 0.033 secs/frame.

Acknowledgments

We would like to thank the anonymous reviewers for their comments. This project is partially supported by the EC Marie Curie International Reintegration Grant (IRG) 223819.

7. REFERENCES

[1] D. Bader, V. Agarwal, and K. Madduri. On the design and analysis of irregular algorithms on the cell processor: A case study of list ranking. In *Proceedings of the 2007 International Parallel and Distributed Processing Symposium (IPDPS 2007)*, page 76, 2007.

[2] H. Baik, K.-H. Sihn, Y.-I. Kim, S. Bae, N. Han, and H. Song. Analysis and parallelization of h.264 decoder on cell broadband engine architecture. In *Proceedings of the 2007 IEEE International Symposium on Signal Processing and Information Technology*, pages 791–795, December 2007.

[3] G. Buehrer, S. Parthasarathy, and M. Goyder. Data mining on the cell broadband engine. In *Proceedings of the 22nd Annual International Conference on Supercomputing (ICS '08)*, pages 26–35, June 2008.

[4] A. Eichenberger, J. OŠBrien, K. OŠBrien, P. Wu, T. Chen, P. Oden, D. Prener, J. Shepherd, B. So, Z. Sura, A. Wang, Z. Zhang, P. Zhao, M. Gschwind, R. Archambault, Y. Gao, and K. R. Using advanced compiler technology to exploit the performance of the cell broadband engine architecture. *IBM Systems Journal*, 45(1):59–84, January 2006.

[5] N. Greene and P. Heckbert. Creating raster omnimax images from multiple perspective views using the elliptical weighted average filter. *IEEE Computer Graphics and Applications*, 6(6):21–27, June 1986.

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

[7] J. Kahle, M. Day, H. Hofstee, C. Johns, T. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604, July/September 2005.

[8] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Implicit and explicit optimizations for stencil computations. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness (MSPC-06)*, pages 51–60, October 2006.

[9] R. Keyes. Cubic convolution interpolation for digital image processing. In *IEEE Transactions on Acoustics, Speech ,and Signal Processing*, volume ASSP-29, December 1981.

[10] M. Knaup, S. Steckmann, O. Bockenbach, and M. Kachelrieß. Tomographic image reconstruction using the cell broadband engine (cbe) general purpose hardware. In *Proceedings Electronic Imaging, Computational Imaging V, SPIE*, volume 6498, 64980, pages 1–10, January 2007.

[11] Y. Liu, H. Jones, S. Vaidya, M. Perrone, B. Tydlitat, and A. Nanda. Speech recognition systems on the cell broadband engine processor. *IBM Journal of Research and Development*, 51(5):583–591, September 2007.

[12] F. Petrini, G. Fossum, J. Fernandez, A. 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.

[13] V. Sachdeva, M. Kistler, E. Speight, and T.-H. Tzang. Exploring the viability of the cell broadband engine for bioinformatics applications. In *Proceedings of the 2007 IEEE International Workshop on High Performance Computational Biology (HiCOMB '07)*, pages 1–8, March 2007.

[14] D. Scarpazza, O. Villa, and F. Petrini. Peak-performance dfa-based string matching on the cell processor. In *Proceedings of the 3rd IEEE/ACM Intl. Workshop on System Management Techniques, Processes, and Services (SMTPS 2007)*, pages 1–8, March 2007.

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

[16] Telerobotics. Omniview motionless camera surveillance system. US Patent Trademark Office, US5359363, February 1993.

[17] 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*, pages 9–20, 2006.

[18] D. Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *Proceedings of the 2000 International Parallel and Distributed Processing Symposium (IPDPS 2000)*, pages 171–180, May 2000.

[19] T. Yamamoto and M. Doi. Design and implementation of panoramic movie system by using commodity 3d graphics hardware. In *Computer Graphics International (CGI) Tokyo, Japan*, pages 14–19, July 2003.

[20] Y. Zhao and K. Kennedy. Dependence-based code generation for a cell processor. In *Proceedings of the 19th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 64–79, November 2006.

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