Projective Transform on Cell: A Case Study

Sharon M. Sacco, Hahn Kim, Sanjeev Mohindra, Peter Boettcher, Chris Bowen,
Nadya Bliss, Glenn Schrader, and Jeremy Kepner
MIT Lincoln Laboratory
{ssacco, hgk, smohindra, boettcher, cbowen, nt, gschrad, kepner}@ll.mit.edu

Abstract

DoD optical applications typically use mobile cameras. In these systems a single image from a camera will be shifted, rotated, tilted, or scaled when compared to the previous image. Image processing can standardize the view of a sequence of images allowing further feature extraction. Here we discuss the development of one correction method, projective transform, for the Sony/Toshiba/IBM Cell processor.

Projective Transform

Projective transform is a powerful tool for standardizing images. Projective transforms are homogeneous warp transforms. The key feature of the projective transform is that straight lines in the source image remain straight lines in the processed image. [1]

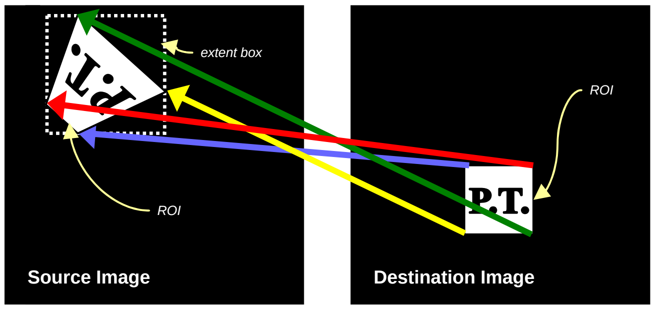

Fig. 1: Projective transform removes distortions from an image and orients the image

Each pixel in the destination image is mapped to a location in the source image. The source location is typically not on a pixel position. A group of pixels around the source position are gathered, and an interpolation is performed to find the pixel value in the destination image.

Parallel Approach

For mobile cameras, the warp coefficients will typically vary from image to image. For each output pixel in the destination image, the corresponding source image pixels will change from image to image. This dynamic mapping does not fit the classic data reorganization patterns.

The Cell architecture requires that all data necessary for a calculation be efficiently loaded into the local store. The distribution chosen for the projective transpose problem is a block distribution of the output image. For each output block, an extent box from the source image can be calculated and its content preloaded. Typically, adjacent blocks have overlapping source extent boxes. Minimizing the edges of the output distribution will reduce the overlap and minimize the number of data transfers.

For communications and data transfers between the PPE and the SPE, we chose Mercury's MCF library. This library encapsulates standard parallel communications, synchronization, and data reorganization as well as DMA functions. We were able to apply data reorganization methods to the output image. The corresponding input image blocks used DMA functions based on pre-computed extent boxes.

Computational kernels were encapsulated into functions. This allowed easy upgrades for optimization purposes since new versions could be written with the same API. In addition, it was easy to verify the accuracy of the new versions as the test images could be compared to the original version.

Compiler Lessons with ANSI C Code

The Cell processor has been supported by two compiler families since its introduction, GNU's gcc/g++ and IBM's XLC C/C++. Both support separate versions for the PPU and SPU. Both PPU compilers are now supporting 64-bit addressing which is a change from last year.

Our concentration has been on the SPU compilers, spu-gcc (4.0.2) and spuxlc (0.8.1) since they are responsible for most of the performance of the Cell. Here we find significant differences between the two compilers.

The performance of the projective transform was significantly affected by the compiler selection. Typical reports give spuxlc over spu-gcc performance improvement as percentages less than 100%. The projective transform had a 3x speedup for standard ANSI C code when built with XLC rather than gcc. The assembler output from spuxlc did not attempt to use SIMD calculations for this application.

Computational accuracy is another concern. Sony / Toshiba / IBM limit the rounding mode for 32-bit floating point on the SPU to truncation. This poses computational challenges if the goal is to duplicate IEEE 754 results obtained from other processors such as x86 and PowerPC. We found that spuxlc handled some rounding difficulties better than spu-gcc.

This work is sponsored by AFRL under Air Force contract FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and not necessarily endorse by the United States Government.

SIMD Coding Experiences

A vectorizing compiler is clearly the simplest way to use vector processors such as the Cell. This is not always an option for all compilers, and even if it is supported, it might not be easy to use or provide sufficient performance.

A more reliable option is for the programmer to use SIMD C extensions for the vector unit. These instructions are very close to the underlying assembly language. This style of programming can produce performance that rivals hand assembly coding for a skilled programmer.

Both spu-gcc and spuxlc were used to compile the computational code for the projective transform. The difference between the two compilers is largely style and convenience. spu-gcc is more user friendly in terms of style than spuxlc. spu-gcc has added C/C++ like binary operators for simple arithmetic. It is also more lenient about typecasting.

SIMD coding also places more algorithmic burden on the programmer. Standard mathematical functions that C/C++ users are accustomed to may not be supported as vector extension. In particular, there is no SIMD support for divide. This presents a challenge for the programmer, particularly with the rounding mode.

Optimizing with MCF

In addition to optimizing the computations, communication optimizations are also needed. Maximizing the destination block size reduced overhead and improved performance as expected.

While MCF tile channels can have multiple components, for small amounts of data, such as the extent box coordinates, performance was improved by combining the data into a single structure to transfer. Using asynchronous DMAs allowed communications and computations to overlap.

Performance Results

Table 1 contains the results of timing for both ANSI C and SIMD C code using either spu-gcc or spuxlc. In all cases, the communications code is identical.

	GCC	XLC
ANSI C	1.42	4.69
SIMD C	21.0	22.7

Table 1: Some typical performance measurements (GOPS) for projective transpose on 8 SPEs

Again there is roughly a 3x speedup from gcc to XLC for ANSI C in this application. If only a small speedup is required for the application's time budget, changing to XLC may be enough. If more performance is needed, SIMD C code gives improvement. Since it is close to assembly code, the difference in performance between the two compilers is small. The programmer has more responsibility for scheduling once SIMD C is used. This particular code was written with many optimization techniques, but not at the level of highly optimized hand coded assembly code.

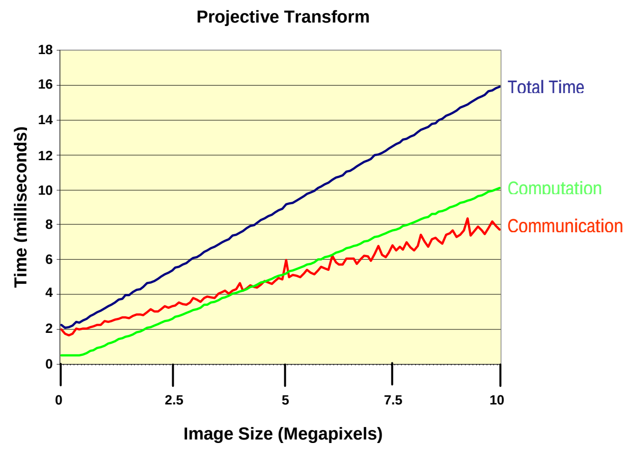

Fig. 2 Times for Projective Transform

Fig. 2 shows the time to perform projective transform as a function of megapixels. The projective transform inner loops were timed and accumulated for the computation. The communication time is the projective transform without the computations. It includes start up overhead as well as memory allocation. The block size is 64 x 64 for this case.

This graph shows linear behavior in all times over the image sizes. We do not have perfect overlapping of computation and communication at this point. Future investigations will seek to improve the performance as needed. From this graph we know that there needs to be changes to lower overhead, increase the computation and communication overlap, and improve the computational performance. The time budget of the application as well as the image size will dictate where we put our efforts.

Conclusions

Programming multi-core processors is not easy. The programmer accustomed to traditional Von Neumann architectures experiences a big increase in the coding complexity. Understanding the tools, such as compilers, and their capabilities is required. Using libraries such as Mercury's MCF can make it easier to program. Even with the available tools, high performance programming still takes more effort than in previous architectures.

References

[1] Many descriptions of projective transform are available on the internet. For example see

http://en.wikipedia.org/wiki/Projective_transformation

[2] International Business Machines Corp, Sony Computer Entertainment Corp., and Toshiba Corp., "Synergistic Processor Unit Instruction Set Architecture", v1.11, © 2006.

[3] International Business Machines Corp, Sony Computer Entertainment Corp., and Toshiba Corp., "C/C++ Language Extensions for Cell Broadband Engine Architecture", v2.2.1, © 2006.