Projective Transform on Cell: A Case Study

Sharon Sacco, Hahn Kim, Sanjeev Mohindra, Peter Boettcher, Chris Bowen, Nadya Bliss, Glenn Schrader and Jeremy Kepner

HPEC 2007

19 September 2007

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

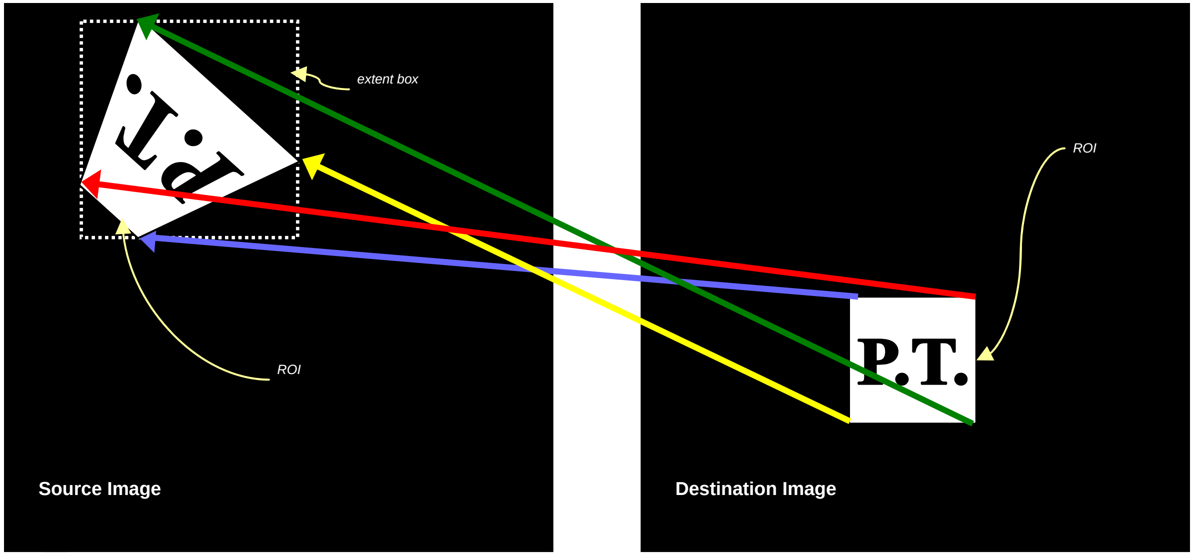
Outline

- **Overview**
  - Why Projective Transform?
  - Projective Transform
  - Cell Features
- Approach
- Coding Tour
- Results
- Summary

Why Projective Transform?

- Aerial surveillance is increasingly important to DoD
- Video / Image understanding needs image processing
- Projective transform is a key image processing kernel

Projective Transform



- Projective Transform is a specialized Warp Transform
  - Performs zoom, rotate, translate, and keystone warping
  - Straight lines are preserved
- Projective Transform registers images from airborne cameras
  - Position of the camera determines the coefficients of the warp matrix

Cell Features

Element Interconnect Bus

- 4 ring buses
- ½ processor speed
- Each ring 16 bytes wide
- Max bandwidth 96 bytes / cycle (204.8 GB/s @ 3.2 GHz)

Synergistic Processing Element

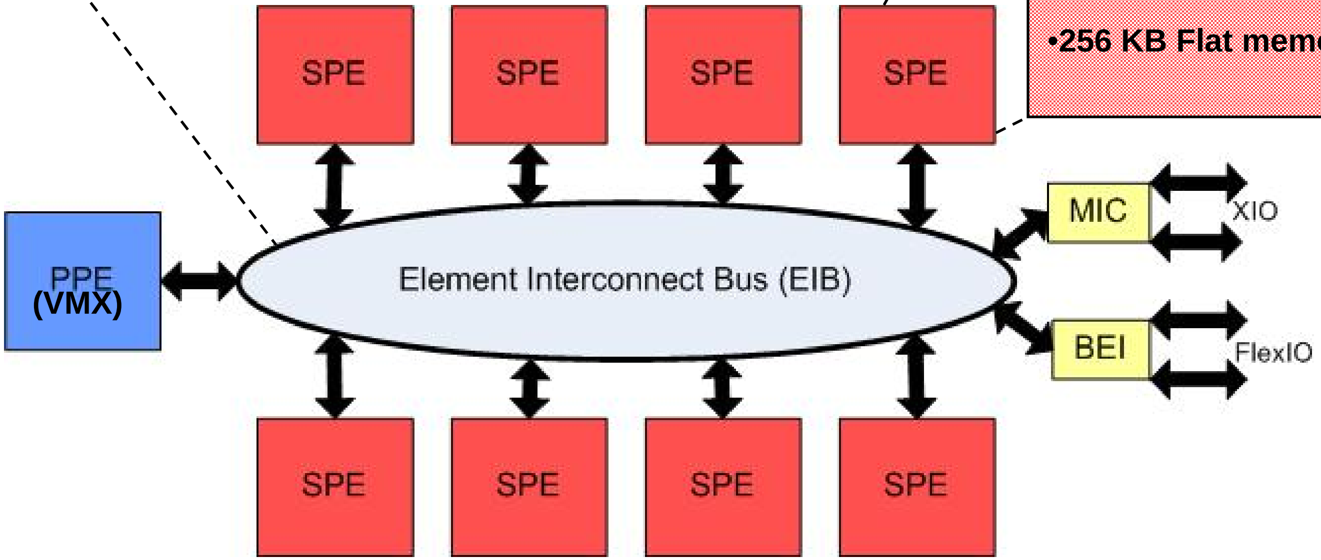
- 128 SIMD Registers, 128 bits wide
- Dual issue instructions

Local Store

- 256 KB Flat memory

Memory Flow Controller

- Built in DMA Engine



Cell's design for games should make it a good image processing processor

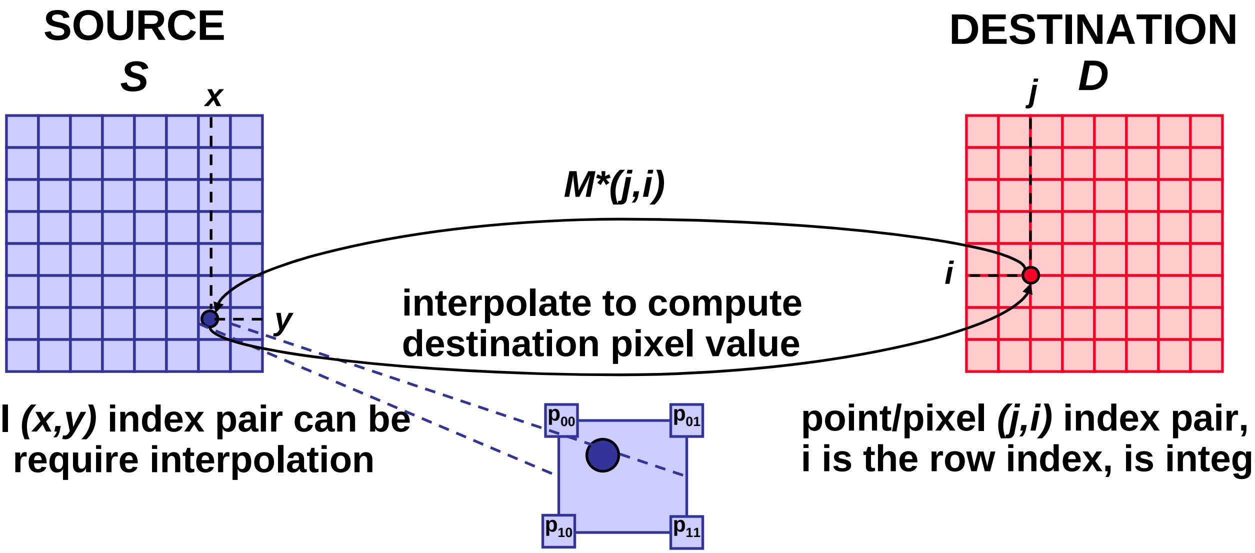
•Overall Performance

- Peak FLOPS @ 3.2 GHz: 204.8 GFLOPS (single), 14.6 GFLOPS (double)
- Processor to Memory bandwidth: 25.6 GB/s
- Cell gives ~2 GFLOPS / W
- Power usage: ~100 W (estimated)

Outline

- Overview
- **Approach**
  - Preliminary Analysis
  - Parallel Approach
  - Cell System
  - Mercury MCF
- Coding Tour
- Results
- Summary

Preliminary Analysis



point/pixel *(x,y)* index pair can be fractional, require interpolation

point/pixel *(j,i)* index pair, where i is the row index, is integer

Transform

$$M = \begin{bmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{bmatrix}$$

$$\begin{bmatrix} x_h \\ y_h \\ w \end{bmatrix} = M * \begin{bmatrix} j \\ i \\ 1 \end{bmatrix} = \begin{bmatrix} m_{00}j + m_{01}i + m_{02} \\ m_{10}j + m_{11}i + m_{12} \\ m_{20}j + m_{21}i + m_{22} \end{bmatrix}$$

$$\left.\begin{array}{l}\text{9 multiplies} \\ \text{6 adds}\end{array}\right\} \text{15 OP}$$

Non-homogeneous Coordinates

$$x = \frac{x_h}{w} = \frac{m_{00}j + m_{01}i + m_{02}}{m_{20}j + m_{21i} + m_{22}}$$

$$y = \frac{y_h}{w} = \frac{m_{10}j + m_{11}i + m_{12}}{m_{20}j + m_{21}i + m_{22}}$$

2 divisions = 2*4 = 8 OP
Cell: 1 division = 4 OP

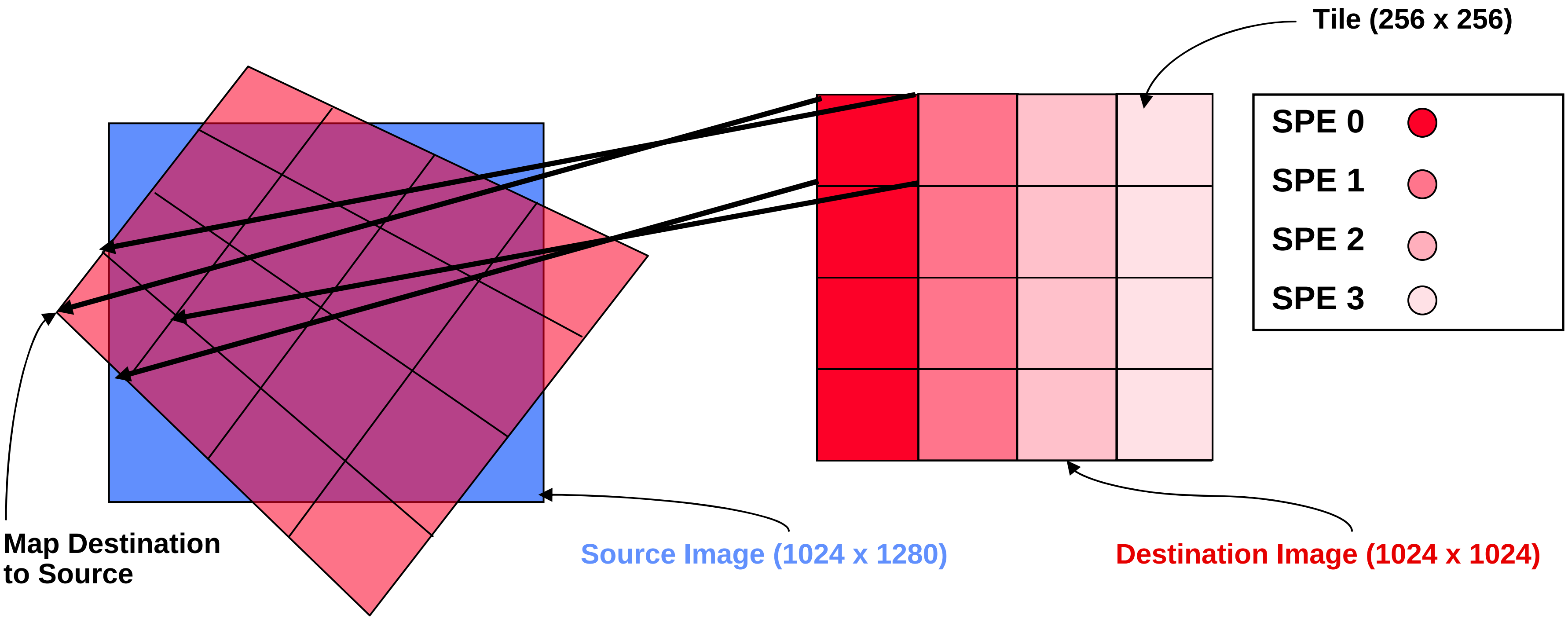
Interpolation

$$V(j,i) = (1-y)*((1-x)*p_{00} + x*p_{01}) + y*((1-x)*p_{10} + x*p_{11})$$

$$\left.\begin{array}{l}\text{6 multiplies} \\ \text{6 adds}\end{array}\right\} \text{12 OP}$$

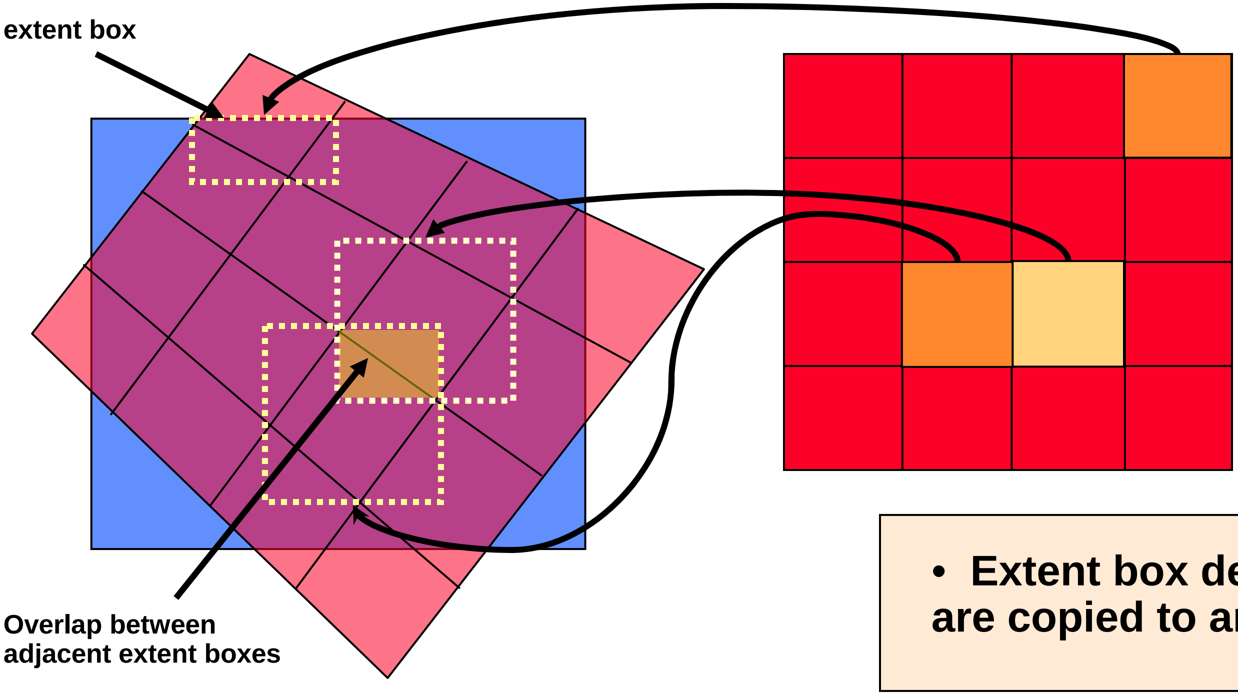
Op count to compute
1 pixel value: 35
Complexity: O(n)

Parallel Approach



- The output image is partitioned into tiles
- Each tile is mapped onto the input image
- Tiles in the output image are partitioned onto SPEs
    - Tiles are distributed “round robin”

Parallel Approach



- Performance is improved by processing whole and partial blocks in code separately

- Extent box determines the pixels that are copied to an SPE’s local store

- For each tile an extent box is calculated for loading into the local store
  - Extent box cannot extend outside of source image
  - Sizes of extent boxes vary within images as well as between images
  - Irregular overlaps between adjacent boxes prevent reuse of data

Mercury Cell Processor Test System

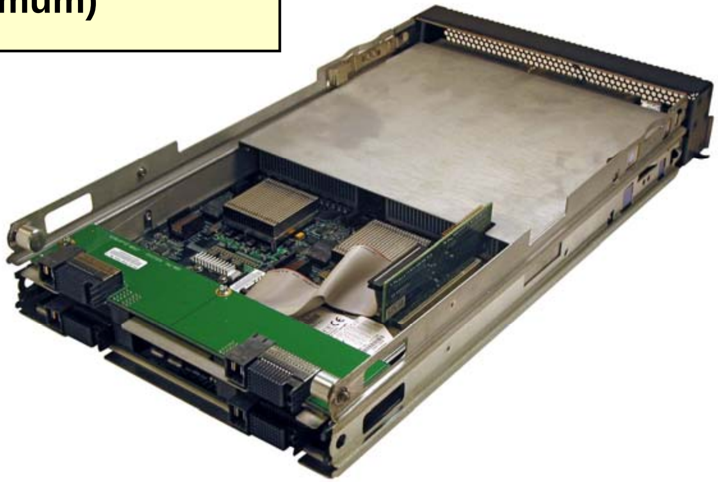
Mercury Cell Processor System

- Single Dual Cell Blade
  - Native tool chain
  - Two 3.2 GHz Cells running in SMP mode
  - Terra Soft Yellow Dog Linux 2.6.17
- Received 03/21/06
  - Booted & running same day
  - Integrated/w LL network < 1 wk
  - Octave (Matlab clone) running
  - Parallel VSIPL++ compiled
- Upgraded to 3.2 GHz December, 2006

• Each Cell has 205 GFLOPS (single precision)
– 410 for system @ 3.2 GHz (maximum)

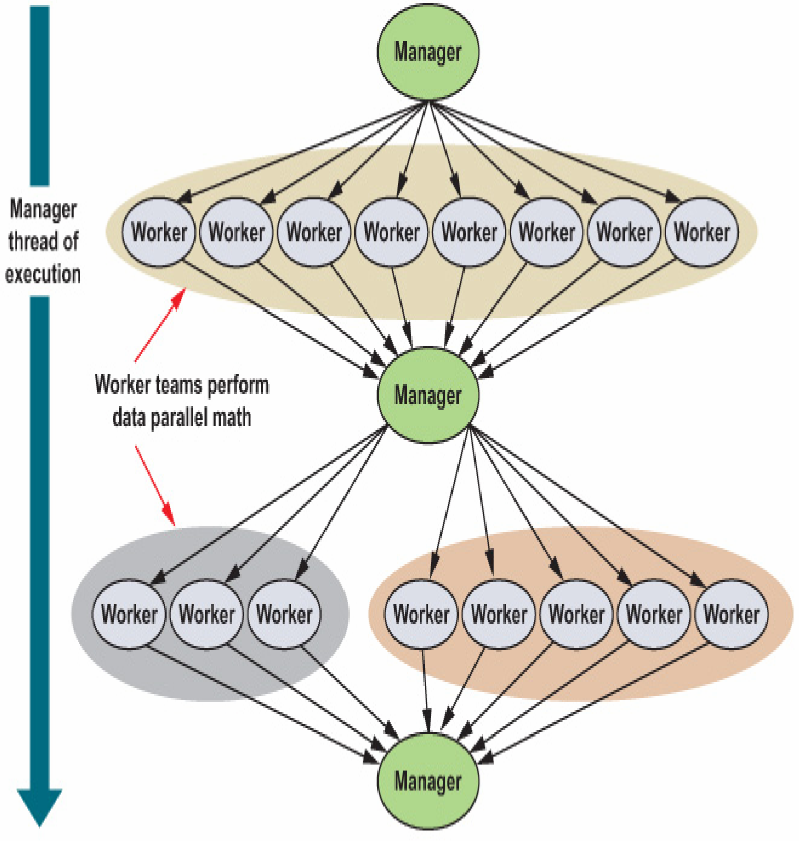
Software includes:

- IBM Software Development Kit (SDK)
  - Includes example programs
- Mercury Software Tools
  - MultiCore Framework (MCF)
  - Scientific Algorithms Library (SAL)
  - Trace Analysis Tool and Library (TATL)



Mercury MCF

- MultiCore Frameworks (MCF) manages multi-SPE programming
  - Function offload engine model
  - Stripmining
  - Intraprocessor communications
  - Overlays
  - Profiling
- Tile Channels expect regular tiles accessed in prescribed ordered
  - Tile channels are good for many common memory access patterns
- Irregular memory access requires explicit DMA transfers



- Leveraging vendor libraries reduces development time
  - Provides optimization
  - Less debugging of application

Outline

- Overview
- Approach
- **Coding Tour**
  - **Manager Communication Code**
  - **Worker Communication Code**
  - SPE Computational Code
- Results
- Summary

PPE Manager Communications

- **Manager responsibilities**
  - Allocate SPEs
  - Manage higher level memory
  - Notify SPEs data is ready
  - Wait for SPEs to release data
  - Initiate clean up

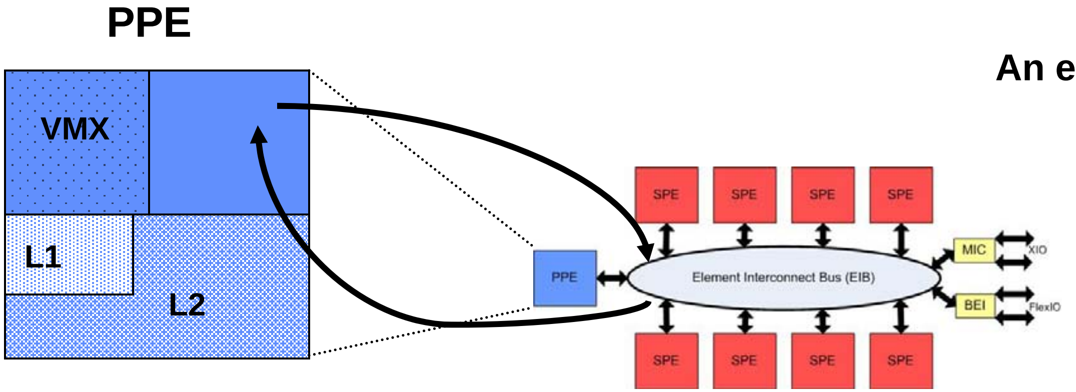
```
rc = mcf_m_tile_channel_put_buffer(h_net,
                                   h_channel_extbox,
                                   &buf_desc_extbox,
                                   MCF_WAIT,
                                   NULL);

rc = mcf_m_tile_channel_get_buffer(h_net,
                                   h_channel_dst,
                                   &buf_desc_dst,
                                   MCF_WAIT,
                                   NULL);

// Disconnect tile channels
rc = mcf_m_tile_channel_disconnect(h_net,
                                   h_channel_extbox,
                                   MCF_WAIT);
```

An excerpt from manager code

- **MCF Tile channel programs are data driven**

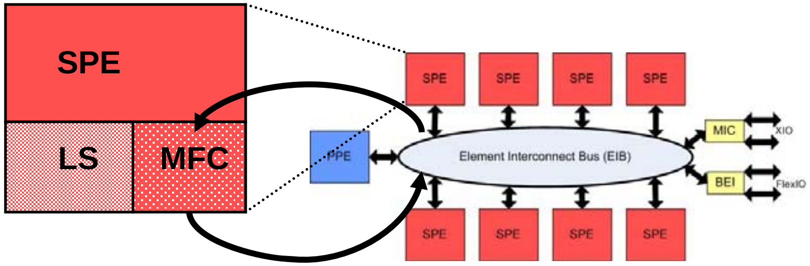


- **Manager communicates with SPEs via EIB**

SPE Worker Communications

- SPE Communication Code
  - Allocates local memory
  - Initiates data transfers to and from XDR memory
  - Waits for transfers to complete
  - Calls computational code

- **SPE communications code manages strip mining of XDR memory**



```
 while (mcf_w_tile_channel_is_not_end_of_frame(h_channel_dst))
 {
// Get a destination image block
   rc = mcf_w_tile_channel_get_buffer(h_channel_dst, &buf_desc_dst,
                                    MCF_RESERVED_FLAG, NULL);

   // If this is the first tile to be processed, then fill the DMA queue
   // Wait for the right dma to complete
   rc = mcf_w_dma_wait(dma_tag,MCF_WAIT);
// Call projective transform kernel
  if (ispartial[dma_tag])
   {  // Process a partial block
     ptInterpolateBlockPart(
                      (unsigned short*) alloc_desc_src[dma_tag]->pp_buffer[0],
                      (unsigned short*) buf_desc_dst->pp_buffer[0],
                      eb_src[dma_tag].x0, eb_src[dma_tag].y0,
                     &eb_dst[dma_tag], coeffs, src_sizeX-1, src_sizeY-1);
  }
   else
   {   // Process a whole block
     ptInterpolateBlock(
                  (unsigned short*) (alloc_desc_src[dma_tag]->pp_buffer[0]),
                   (unsigned short int*) buf_desc_dst->pp_buffer[0],
                   eb_src[dma_tag].x0, eb_src[dma_tag].y0,
                  &eb_dst[dma_tag], coeffs);

…       // load next extent box contents and other operations

      rc = mcf_w_tile_channel_put_buffer(h_channel_dst,
                                    &buf_desc_dst, MCF_RESERVED_FLAG,
                                    NULL);
```

An excerpt from worker code

Outline

- Overview
- Approach
- **Coding Tour**
  - Manager Communication Code
  - Worker Communication Code
  - **SPE Computational Code**
- Results
- Summary

Reference C

- **C is a good start for code design**
  - **Speed not important**

Find precise position in image

Find upper left pixel and offsets

Estimate pixel value using bi-linear interpolation

```
t1 = fI * coeffs[2][1] + coeffs[2][2];
t2 = fI * coeffs[0][1] + coeffs[0][2];
t3 = fI * coeffs[1][1] + coeffs[1][2];

for (j = min_j, fJ = (float)min_j; j <= max_j; j++,
      fJ += 1.0){
   // Find position in source image
   df = 1.0 / (fJ * coeffs[2][0] + t1);
   xf = (fJ * coeffs[0][0] + t2) * df;
   yf = (fJ * coeffs[1][0] + t3) * df;

   // Find base pixel address and offsets
   x = (int) xf;
   y = (int) yf;
   dx = (int)(256.0 * (xf - x));
   dy = (int)(256.0 * (yf - y));

   // Pick up surrounding pixels, bilinear interpolation
   s = &srcBuffer[y - yOffset][x - xOffset];
   rd = *s * (256 - dx) + *(s + 1) * dx;
   s += BLOCKSIZE << 1;
   yr = *s * (256 - dx) + *(s + 1) * dx;
   rd = rd * (256 - dy) + yr * dy;
   *ptrRunning = rd >> 16;   //  Write to des. image
   ptrRunning++;
```

Computational Code for Row in Whole Tile in ANSI C

C with SIMD Extensions

- **SIMD C is more complicated than ANSI C**
  - **Does not follow same order**
- **SPE only sees local store memory**

```
// Pick up surrounding pixels, bilinear interpolation
s = &srcBuffer[y - yOffset][x - xOffset];
rd = *s * (256 - dx) + *(s + 1) * dx;
s += BLOCKSIZE << 1;

yr = *s * (256 - dx) + *(s + 1) * dx;

rd = rd * (256 - dy) + yr * dy;
```

Bi-linear Interpolation from ANSI C Version

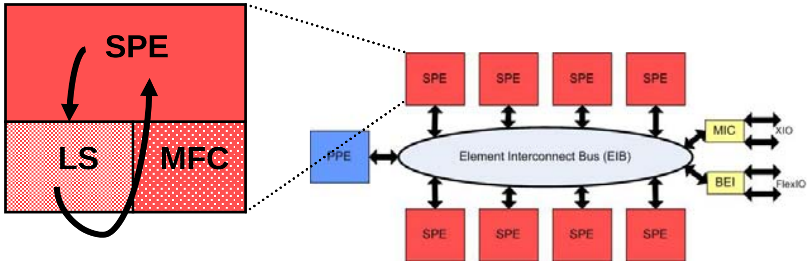
```
sptr = (unsigned short *)spu_extract(y2,0);
s1 = *sptr;

yr = spu_add(spu_mulo((vector unsigned short)LL,
             (vector unsigned short)xdiff),
         spu_mulo((vector unsigned short)LR,
             (vector unsigned short)dx1));

s2 = *(sptr + 1);
s3 = *(sptr + si_to_int((qword)twoBlocksize));

rd1 = spu_add(
        spu_add(
          spu_add(
            spu_mulo((vector unsigned short)rd1,
                  (vector unsigned short)ydiff),
            (vector unsigned int)spu_mulh(
                  (vector signed short)rd1,
                  (vector signed short)ydiff)),
          spu_add((vector unsigned int)spu_mulh(
                  (vector signed short)ydiff,
                  (vector signed short)rd1),
            spu_mulo((vector unsigned short)yr,
                (vector unsigned short)dy1))),
          spu_add((vector unsigned int)spu_mulh(
                (vector signed short)yr,
                (vector signed short)dy1),
            (vector unsigned int)spu_mulh(
                 (vector signed short)dy1,
                 (vector signed short)yr)));
```

An excerpt from SIMD version of Projective Transform



Rounding and Division

```
df = 1.0 / (fJ * coeffs[2][0] + t1);
 xf = (fJ * coeffs[0][0] + t2) * df;
 yf = (fJ * coeffs[1][0] + t3) * df;

 x = (int) xf;  // Note that next step is "float to fix"
 y = (int) yf;
```

ANSI C Implementation

```
//df = vector float(1.0) / (fJ * vector float(*(coeffs + 6)) + T1);

    yf = spu_madd(fJ, spu_splats(*(coeffs + 6)), T1);
    df = spu_re(yf);            // y1 ~ (1 / x), 12 bit accuracy
    yf = spu_nmsub(yf, df, f1); // t1 = -(x * y1 - 1.0)
    df = spu_madd(yf, df, df);
                                // y2 = t1 * y1 + y1, done with
                                // Newton Raphson

    xf = spu_madd(fJ, spu_splats(*coeffs), T2);
    yf = spu_madd(fJ, spu_splats(*(coeffs + 3)), T3);
    xf = spu_mul(xf, df);
    yf = spu_mul(yf, df);

    // nudge values up to compensate for truncation
    xf = (vector float)spu_add((vector unsigned int) xf, 1);
    yf = (vector float)spu_add((vector unsigned int) yf, 1);
```

SIMD C Implementation with Minimal Correction

- **Division takes extra steps**
- **Data range and size may allow shortcuts**
- **Expect compiler dependent results**

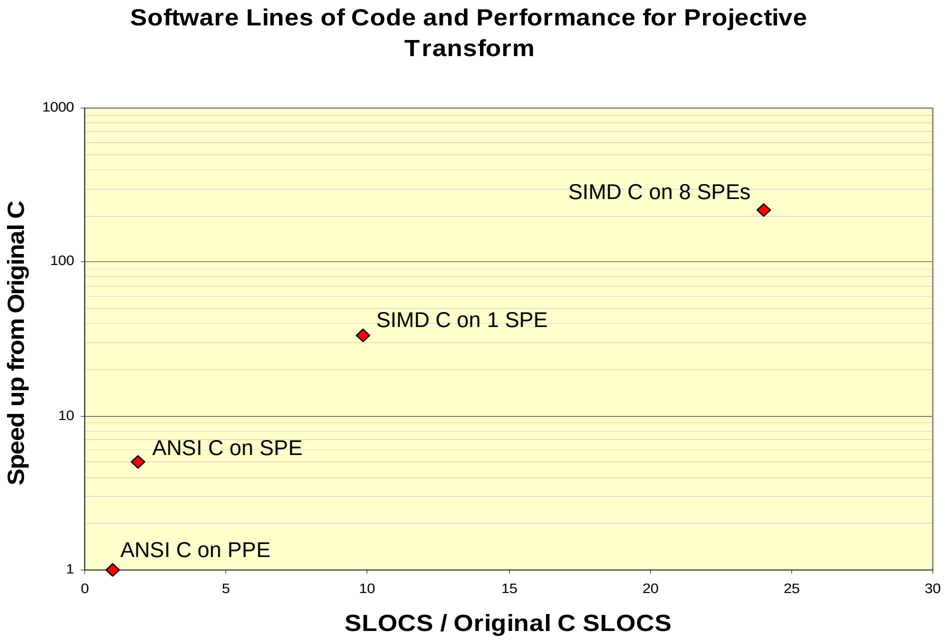
- **Truncation forces some changes in special algorithms for accuracy**

Outline

- Overview
- Approach
- Coding Tour
- **Results**
  - SLOCs and Coding Performance
  - Compiler Performance
  - Covering Data Transfers
- Summary

SLOCs and Coding Performance

	SLOCS	GOPS (10 M pix)
ANSI C (PPE)	52	0.126
ANSI C (SPE)	97	0.629
SIMD C	512	4.20
Parallel SIMD	1248	27.41



- **Clear tradeoff between performance and effort**
  - **C code simple, poor performance**
  - **SIMD C, more complex to code, reasonable performance**

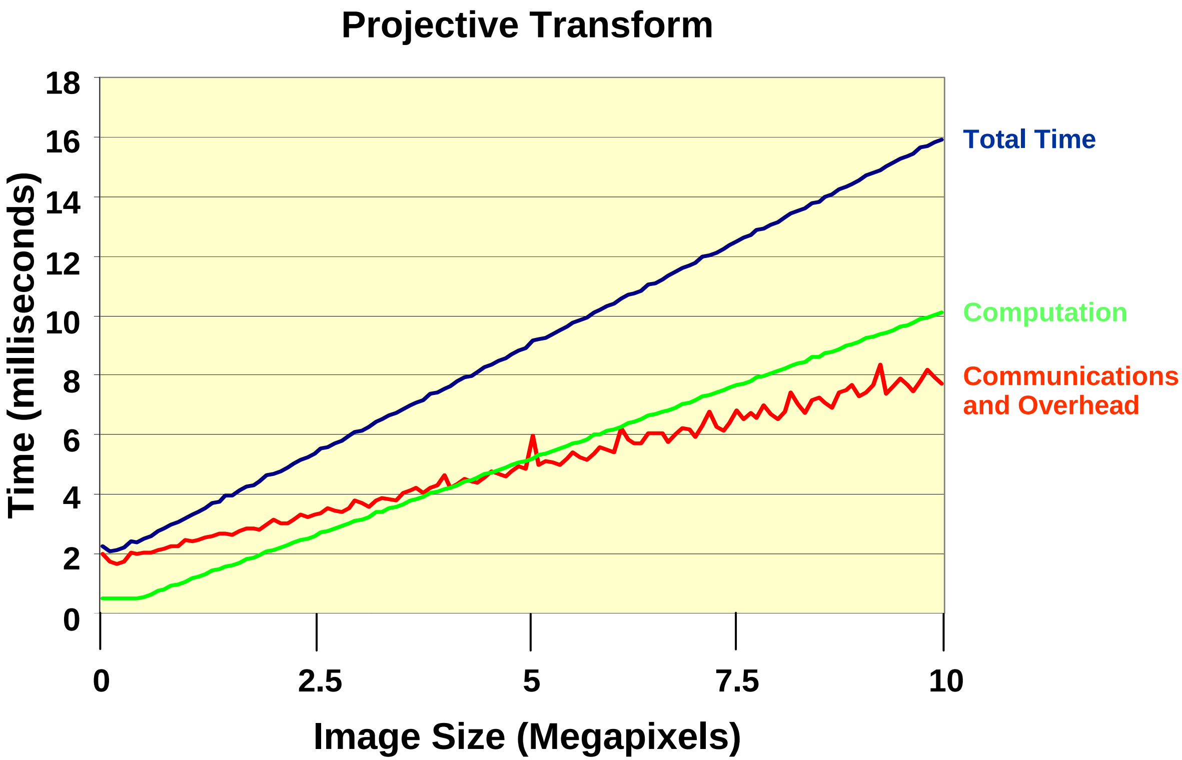
Compiler Performance

- GOPS (giga operations per second) based on 40 operations / pixel
- 1 SPE used
- Compiler switches vary, but basic level of optimization is the same (-O2)
- Performance will vary by image size (10 M pixel image used)
- XLC only used on SPE code

	ANSI C	SIMD C
GCC / G++ (v. 4.1.1) (GOPS)	0.182	3.68
XLC (v. 8.01) (GOPS)	0.629	4.20
XLC / GCC	3.46	1.14

- **XLC outperforms GCC / G++ on SPEs**
  - **Significant improvement for serial ANSI C code**
  - **Some improvement with SIMD code**

Covering Data Transfers



- 8 SPEs are used
- About 2 msec overhead
- Computation dominates
  - Assembly code would be the next optimization if needed
- Communications are partially covered by computations

- Timing for projective transform scales with image size

Summary

- Good Cell programming takes work
  - Compiler choice can noticeably affect performance, particularly if ANSI C is used
  - SIMD C/C++ extensions perform much better than ANSI C/C++, but at the price of code complexity
  - Middleware such as Mercury's MCF makes coding easier
  - Rounding mode on SPEs presents challenges to users

- Better middleware will make programming easier for users
  - There needs to be a level of programming where the user does not have to become a Cell expert

Backup

The Plan

OP Count Assumptions:
Transform: 3 mults + 3 adds = 6 OPs
Total op count: 6+12+8 = 26 OPs/pixel

Total operation count requirement/second:

- 26 OPs/pixel * 11,000,000 pixels/frame * 4 frames = 1,144,000,000 OPS = 1.144 gigaOPS

1 SPE processing capability:

- 25.6 GFLOPS

Time complexity calculation assumptions:

- Each pixel is 16 bits or 2 bytes
- 1 SPE
- Sub-image size conducive to double-buffering
- Double buffering is not used

(Assume that operations on 2 byte integers cost the same as operations on single precision, 4 byte, floating point numbers)

- **Estimating the algorithm and communication requirements helps to predict performance**

Local Store (LS) = 256 KB
Assume 80KB dedicated to MCF and other code

- 256 - 80 = 176 KB for data

Allow another 20% space for incidentals

- 176 KB * 0.8 = 140.8 KB for data
- 140.8 KB * 1024 = 144,180 bytes

Number of pixel that fit into LS

- 144,180 bytes / (2 bytes/pixel) = 72,090 pixels

Need to store both source and destination sub-image
(For 1 unit of destination space, need 4 units of source)

- 72,090 pixels / (1+4) = 14,418 pixels of destination can be computed on a single SPE

Setup for double buffering

- 14,418/2 ~= 7,000 pixels can be computed in LS

To compute each pixel, need to transfer in source (4*7000 pixels*2 bytes/pixel) and transfer out the destination (7000 pixels*2 bytes/pixel)

To compute 7,000 pixels in the destination, have to transfer (5*7000*2) = 70,000 bytes

Time complexity of data transfer (ignore latency) at 25.6 GB/s
70,000 bytes/$25.6*10^9$ bytes/sec = $2.73*10^{-6}$ sec
Time complexity of computation at 25.6 GFLOPS

- (7,000 pixels * 26 OP/pixel)/$25.6*10^9$FLOPS = $7.11*10^{-6}$

Number of 7000 pixel blocks in 11MPixel image
11,000,000/7,000 = 1572
Time complexity of computing 4 frames

- 4 frames * 1572 blocks *($2.73*10^{-6}+7.11*10^{-6}$) = 0.0620 sec

Preliminary estimate of resources needed for Projective Transform