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

### Overall Performance

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

### Cell Features Diagram

- **Synergistic Processing Element**
  - 128 SIMD Registers, 128 bits wide
  - Dual issue instructions

- **Local Store**
  - 256 KB Flat memory
  - Built in DMA Engine

- **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)

- **Memory Flow Controller**

- **Element Interconnect Bus (EIB)**

- **Cell’s design for games should make it a good image processing processor**
Outline

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

Transform

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

\[
\begin{bmatrix}
x \\
y \\
w
\end{bmatrix} = \begin{bmatrix}
m_{00} & m_{01} & m_{02} \\
m_{10} & m_{11} & m_{12} \\
m_{20} & m_{21} & m_{22}
\end{bmatrix} \begin{bmatrix}
x_k \\
y_k \\
w
\end{bmatrix}
\]

9 multiplies
6 adds \( \rightarrow \) 15 OP

Non-homogeneous Coordinates

\[
x = \frac{x_k}{w} = \frac{m_{00}j + m_{01}i + m_{02}}{m_{20}j + m_{21}i + m_{22}}
\]

\[
y = \frac{y_k}{w} = \frac{m_{10}j + m_{11}i + m_{12}}{m_{20}j + m_{21}i + m_{22}}
\]

2 divisions = 2*4 = 8 OP

Cell: 1 division = 4 OP

Interpolation

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

6 multiplies
6 adds \( \rightarrow \) 12 OP

1 pixel value: 35
Complexity: O(n)

Op count to compute
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

- 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

- 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
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 VSIP++ 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

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

An excerpt from manager code

```c
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);
```

- **Manager communicates with SPEs via EIB**

*MIT Lincoln Laboratory*
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
• 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

Computational Code for Row in Whole Tile in ANSI C

```c
// 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 dest. image
    ptrRunning++;
}
```

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

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

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

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

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

```c
rd1 = spu_add(spu_add(spu_add(spu_add(spu_mulo((vector unsigned short)ydiff, (vector unsigned short)dy1),
                                spu_mulh((vector signed short)ydiff, (vector signed short)rd1)),
                                spu_mulh((vector signed short)rd1, (vector signed short)dy1)),
                                spu_mulo((vector unsigned short)ydiff, (vector unsigned short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)dy1, (vector unsigned short)dy1)),
                                spu_mulh((vector signed short)dy1, (vector signed short)dy1)),
                                spu_mulh((vector signed short)ydiff, (vector signed short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)ydiff, (vector unsigned short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)dy1, (vector unsigned short)dy1)),
                                spu_mulh((vector signed short)dy1, (vector signed short)dy1)),
                                spu_mulh((vector signed short)ydiff, (vector signed short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)ydiff, (vector unsigned short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)dy1, (vector unsigned short)dy1)),
                                spu_mulh((vector signed short)dy1, (vector signed short)dy1)),
                                spu_mulh((vector signed short)ydiff, (vector signed short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)ydiff, (vector unsigned short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)dy1, (vector unsigned short)dy1)),
                                spu_mulh((vector signed short)dy1, (vector signed short)dy1)),
                                spu_mulh((vector signed short)ydiff, (vector signed short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)ydiff, (vector unsigned short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)dy1, (vector unsigned short)dy1)),
                                spu_mulh((vector signed short)dy1, (vector signed short)dy1)),
                                spu_mulh((vector signed short)ydiff, (vector signed short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)ydiff, (vector unsigned short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)dy1, (vector unsigned short)dy1)),
                                spu_mulh((vector signed short)dy1, (vector signed short)dy1)),
                                spu_mulh((vector signed short)ydiff, (vector signed short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)ydiff, (vector unsigned short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)dy1, (vector unsigned short)dy1)),
                                spu_mulh((vector signed short)dy1, (vector signed short)dy1)),
                                spu_mulh((vector signed short)ydiff, (vector signed short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)ydiff, (vector unsigned short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)dy1, (vector unsigned short)dy1)),
                                spu_mulh((vector signed short)dy1, (vector signed short)dy1)),
                                spu_mulh((vector signed short)ydiff, (vector signed short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)ydiff, (vector unsigned short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)dy1, (vector unsigned short)dy1)),
                                spu_mulh((vector signed short)dy1, (vector signed short)dy1)),
                                spu_mulh((vector signed short)ydiff, (vector signed short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)ydiff, (vector unsigned short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)dy1, (vector unsigned short)dy1)),
                                spu_mulh((vector signed short)dy1, (vector signed short)dy1)),
                                spu_mulh((vector signed short)ydiff, (vector signed short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)ydiff, (vector unsigned short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)dy1, (vector unsigned short)dy1)),
                                spu_mulh((vector signed short)dy1, (vector signed short)dy1)),
                                spu_mulh((vector signed short)ydiff, (vector signed short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)ydiff, (vector unsigned short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)dy1, (vector unsigned short)dy1)),
                                spu_mulh((vector signed short)dy1, (vector signed short)dy1)),
                                spu_mulh((vector signed short)ydiff, (vector signed short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)ydiff, (vector unsigned short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)dy1, (vector unsigned short)dy1)),
                                spu_mulh((vector signed short)dy1, (vector signed short)dy1)),
                                spu_mulh((vector signed short)ydiff, (vector signed short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)ydiff, (vector unsigned short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)dy1, (vector unsigned short)dy1)),
                                spu_mulh((vector signed short)dy1, (vector signed short)dy1)),
                                spu_mulh((vector signed short)ydiff, (vector signed short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)ydiff, (vector unsigned short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)dy1, (vector unsigned short)dy1)),
                                spu_mulh((vector signed short)dy1, (vector signed short)dy1)),
                                spu_mulh((vector signed short)ydiff, (vector signed short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)ydiff, (vector unsigned short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)dy1, (vector unsigned short)dy1)),
                                spu_mulh((vector signed short)dy1, (vector signed short)dy1)),
                                spu_mulh((vector signed short)ydiff, (vector signed short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)ydiff, (vector unsigned short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)dy1, (vector unsigned short)dy1)),
                                spu_mulh((vector signed short)dy1, (vector signed short)dy1)),
                                spu_mulh((vector signed short)ydiff, (vector signed short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)ydiff, (vector unsigned short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)dy1, (vector unsigned short)dy1)),
                                spu_mulh((vector signed short)dy1, (vector signed short)dy1)),
                                spu_mulh((vector signed short)ydiff, (vector signed short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)ydiff, (vector unsigned short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1)),
                                spu_mulo((vector unsigned short)dy1, (vector unsigned short)dy1)),
                                spu_mulh((vector signed short)dy1, (vector signed short)dy1)),
                                spu_mulh((vector signed short)ydiff, (vector signed short)ydiff)),
                                spu_mulh((vector signed short)rd1, (vector signed short)rd1))}{
```

Bi-linear Interpolation from ANSI C Version

// 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;
```

An excerpt from SIMD version of Projective Transform
Rounding and Division

**ANSI C Implementation**

\[
\begin{align*}
df &= 1.0 / (fJ * coeffs[2][0] + t1); \\
xf &= (fJ * coeffs[0][0] + t2) * df; \\
yf &= (fJ * coeffs[1][0] + t3) * df; \\
x &= (\text{int}) xf; & \text{// Note that next step is “float to fix”} \\
y &= (\text{int}) yf;
\end{align*}
\]

**SIMD C Implementation with Minimal Correction**

\[
\begin{align*}
//df = \text{vector float}(1.0) / (fJ * \text{vector float}(*\text{coeffs} + 6)) + T1;\\n
yf &= \text{spu_madd}(fJ, \text{spu_splats}(\text{coeffs} + 6), T1); \\
df &= \text{spu_re}(yf); & \text{// y1 ~ (1 / x), 12 bit accuracy} \\
yf &= \text{spu_nmsub}(yf, df, f1); & \text{// t1 = -(x * y1 - 1.0)} \\
df &= \text{spu_madd}(yf, df, df); & \text{// y2 = t1 * y1 + y1, done with} \\
& \text{// Newton Raphson} \\
xf &= \text{spu_madd}(fJ, \text{spu_splats}(\text{coeffs}), T2); \\
yf &= \text{spu_madd}(fJ, \text{spu_splats}(\text{coeffs} + 3), T3); \\
xf &= \text{spu_mul}(xf, df); \\
yf &= \text{spu_mul}(yf, df); \\
// nudge values up to compensate for truncation \\
xf &= (\text{vector float})\text{spu_add}((\text{vector unsigned int}) xf, 1); \\
yf &= (\text{vector float})\text{spu_add}((\text{vector unsigned int}) yf, 1);
\end{align*}
\]

**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

<table>
<thead>
<tr>
<th></th>
<th>SLOCS</th>
<th>GOPS (10 M pix)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSI C (PPE)</td>
<td>52</td>
<td>0.126</td>
</tr>
<tr>
<td>ANSI C (SPE)</td>
<td>97</td>
<td>0.629</td>
</tr>
<tr>
<td>SIMD C</td>
<td>512</td>
<td>4.20</td>
</tr>
<tr>
<td>Parallel SIMD</td>
<td>1248</td>
<td>27.41</td>
</tr>
</tbody>
</table>

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

- GPUs (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

<table>
<thead>
<tr>
<th></th>
<th>ANSI C</th>
<th>SIMD C</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC / G++</td>
<td>0.182</td>
<td>3.68</td>
</tr>
<tr>
<td>(v. 4.1.1) (GOPS)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XLC</td>
<td>0.629</td>
<td>4.20</td>
</tr>
<tr>
<td>(v. 8.01) (GOPS)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XLC / GCC</td>
<td>3.46</td>
<td>1.14</td>
</tr>
</tbody>
</table>

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

Projective Transform

- Timing for projective transform scales with image size

- 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

Image Size (Megapixels)

Time (milliseconds)

Total Time

Computation

Communications and Overhead
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

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

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*10^8 bytes/sec = 2.73*10^-6 sec

Time complexity of computation at 25.6 GFLOPS
• (7,000 pixels * 26 OP/pixel)/25.6*10^8FLOPS = 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

Estimating the algorithm and communication requirements helps to predict performance

Preliminary estimate of resources needed for Projective Transform