First-hand experience on porting MATPHOT code to SRC platform

Volodymyr (Vlad) Kindratenko
NCSA, UIUC
kindr@ncsa.uiuc.edu
Presentation outline

• What is MATPHOT?
• MATPHOT code
• Testbed code
• Implementations on SRC
  – early production 2U dual PIV with 1 GB memory SNAP and dual MAP platform
  – Carte 1.9 development environment
• Lessons learned
• Conclusions
What is MATPHOT?

- **Author**
  - **Kenneth Mighell**, National Optical Astronomy Observatory

- **What it does**
  - Algorithm for Accurate and Precise Stellar Photometry and Astrometry Using Discrete Point Spread (PSF) Functions

- **Where to get it**
  - [http://www.noao.edu/staff/mighell/matphot/](http://www.noao.edu/staff/mighell/matphot/)
What is MATPHOT?

• Simulated observational data
• Best model of the observation

Images are courtesy of Kenneth Mighell from the National Optical Astronomy Observatory

National Center for Supercomputing Applications
MATPHOT code profiling

- MPD - MATPHOT demonstration program
  - Compile and link with \(-p\) option
  - Run as usual
  - Rung \textit{gprof} afterwards

<table>
<thead>
<tr>
<th>% time</th>
<th>cumulative seconds</th>
<th>self seconds</th>
<th>calls</th>
<th>self s/call</th>
<th>total s/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>84.17</td>
<td>82.39</td>
<td>82.39</td>
<td>91504</td>
<td>0</td>
<td>0</td>
<td>mpd_ipImageS_sshift_fnV5_VK</td>
</tr>
<tr>
<td>7.49</td>
<td>89.72</td>
<td>7.33</td>
<td>152</td>
<td>0.05</td>
<td>0.59</td>
<td>mpd_ipImageS_Shift2d_fs4</td>
</tr>
<tr>
<td>5.39</td>
<td>95</td>
<td>5.28</td>
<td>152</td>
<td>0.03</td>
<td>0.63</td>
<td>mpd_fdeis0d_fs7</td>
</tr>
<tr>
<td>0.87</td>
<td>95.85</td>
<td>0.85</td>
<td>65</td>
<td>0.01</td>
<td>0.01</td>
<td>mpd_LmmS_AlphaBetaSet_fs1</td>
</tr>
<tr>
<td>0.5</td>
<td>96.34</td>
<td>0.49</td>
<td>152</td>
<td>0</td>
<td>0</td>
<td>mpd_ipImageS_Add_star_fs7</td>
</tr>
<tr>
<td>0.3</td>
<td>96.63</td>
<td>0.29</td>
<td>142</td>
<td>0</td>
<td>0</td>
<td>mpd_fmodel_fs7</td>
</tr>
<tr>
<td>0.15</td>
<td>96.78</td>
<td>0.15</td>
<td></td>
<td></td>
<td></td>
<td>snorm</td>
</tr>
</tbody>
</table>

National Center for Supercomputing Applications
void mdp_iplImageS_sshift_fnV5_VK(float x[], int n, float shift, float hole, float xp[], float sinc[])
{
    /* do the fractional shift (if any) */
    if ( ( fabs(shift) > EPS ) && ( fabs(shift) < 1.0-EPS ) )
    {
        /* convolve the input data with the sinc array */
        for ( point=0; point < n; point++)
            for ( lobe=0, xp[point]=0; lobe < 21; lobe++)
                {npix = point - (lobe-10);
                 if( (npix >= 0) && (npix < n) )
                     xp[point] += sinc[lobe]*x[npix];
                 else
                     xp[point] += sinc[lobe]*hole;}
    } else /* shift is positive */
    {
        for ( i=n-1, point = n-1-ishift; i >= 0; --i)
            {xp[i] = xp[point];}
    }
    else
        for( i=0; i<n; i++) xp[i]=x[i];

    /* do the integer shift, if any */
    if (ishift < 0) /* shift is negative */
    {
        for ( i=0, point = -ishift; i < n; i++, point++)
        {if( point < 0 )
            xp[i]=hole;
        else if( point >= n )
            xp[i]=hole;
        else
            xp[i]=xp[point];}
    }
    else /* shift is positive */
    {
        for ( i=0, point = ishift; i < n; i++, point++)
        {if( point < 0 )
            xp[i]=hole;
        else if( point >= n )
            xp[i]=hole;
        else
            xp[i]=xp[point];}
    }
}
```c
int mpd_ipImageS_Shift2d_fs4(struct
    mpd_ipImageS_s *iPS, double deltaxD, 
    double deltayD, struct mpd_ipImageS_s *oPS)
{
    /* allocate memory for intermediate storage */
    iAF = calloc( nelemT, sizeof(float) );
    oAF = calloc( nelemT, sizeof(float) );

    /* compute shift coefficients */
    init_sinc_array_VK(deltaxF, sincX);

    /* shift DELTAX pixels in the X direction */
    for (iy=0; iy<ny; ++iy)
    {
        for (ix=0; ix<nx; ++ix)
            iAF[ix] = (float)iPS->matrixd[iy][ix];

        mpd_ipImageS_sshift_fnV5_VK ( iAF, nx, 
            deltaxF, zeroF, oAF, sincX );

        for (ix=0; ix<nx; ++ix)
            oPS->matrixd[iy][ix] = oAF[ix];
    }

    /* compute shift coefficients */
    init_sinc_array_VK(deltayF, sincY);

    /* shift DELTAY pixels in the Y direction */
    for (ix=0; ix<nx; ++ix)
    {
        for (iy=0; iy<ny; ++iy)
            iAF[iy] = (float)oPS->matrixd[iy][ix];

        mpd_ipImageS_sshift_fnV5_VK ( iAF, ny, 
            deltayF, zeroF, oAF, sincY );

        for (iy=0; iy<ny; ++iy)
            oPS->matrixd[iy][ix] = oAF[iy];
    }

    free( iAF );
    free( oAF );
}
```
MATPHOT internals

- 1D convolution 21-pixel-wide *damped sinc function*

\[
 f^{\text{shifted}}(x_0) = \sum_{i=-10}^{10} f(x_i) \frac{\sin(\pi(x_i - x_0))}{\pi(x_i - x_0)} \exp\left(-\left[\frac{x_i - x_0}{3.25}\right]^2\right)
\]

original image  \[\rightarrow\]  \[x\text{ shift}\]  \[\rightarrow\]  \[y\text{ shift}\]
Overall algorithm

mpd_ipImageS_Shift2d_fs4

- For each row of pixels
  - copy pixels to a 1D array
  - call _sshift_fnV5_
  - copy results back to image memory

- For each column of pixels
  - copy pixels to a 1D array
  - call _sshift_fnV5_
  - copy results back to image memory

- Is called once per image

mpd_ipImageS_sshift_fnV5_VK

- For each 1D array
  - 21 point convolution
    - 21 multiplications
    - 20 additions

- Is called once per each row and column

National Center for Supercomputing Applications
```c
int main (int argc, char *argv[]) {
    
    /* allocate images per command line arguments */
    
    gettimeofday(&time0, NULL);

    proc_cpu(image1, image2, dx, dy);

    gettimeofday(&time1, NULL);

    /* print time difference */
    
}

int proc_cpu (pgm_image *image_in, pgm_image *image_out, float dx, float dy) {
    
    init_sinc_array_CPU(dx, sinc_x);
    init_sinc_array_CPU(dy, sinc_y);

    /* shift DELTAX pixels in the X direction */
    for (iy = 0; iy < image_in->sn; ++iy) {
        for (ix = 0; ix < image_in->sm; ++ix)
            iAx[ix] = image_in->img[iy*image_in->sm+ix];
        
        sshift(iAx, image_in->sm, dx, zeroF, oAx, sinc_x);
        for (ix = 0; ix < image_out->sm; ++ix)
            image_out->img[iy*image_out->sm+ix] = oAx[ix];
    }

    /* shift DELTAY pixels in the Y direction */
    for (ix = 0; ix < image_in->sm; ++ix) {
        for (iy = 0; iy < image_in->sn; ++iy)
            iAy[iy] = image_out->img[iy*image_in->sm+ix];
        
        sshift(iAy, image_in->sn, dy, zeroF, oAy, sinc_y);
        for (iy = 0; iy < image_out->sn; ++iy)
            image_out->img[iy*image_out->sm+ix] = oAy[iy];
    }

    ...
```
void \texttt{sshift}(float *x, long n, float shift, float hole, float *xp, float *sinc)
{
    ...
    /* do the fractional shift (if any) */
    if ( fshift is significant )
    {
        /* convolve the input data with the sinc array */
        for (point = 0; point < n; point++)
        {
            xp[point] = 0.0f;
            for (lobe = 0; lobe < 21; lobe++)
            {
                npix = point - (lobe - 10);
                if ( (npix >= 0) && (npix < n) )
                    xp[point] += sinc[lobe] * x[npix];
                else
                    xp[point] += sinc[lobe] * hole;
            }
        }
    }
}

void \texttt{init\_sinc\_array\_CPU}(float shift, float *sinc)
{
    int ishift; /* Integer part of shift. */
    float fshift; /* Fractional part of shift. */
    int point; /* This is a counter variable. */
    float y; /* Another counter variable. */

    ishift = (int)shift;
    fshift = shift - ishift;

    /* initialize the sinc array */
    for (y = fshift+10, point = 0 ; point < 21; --y, point++)
    {
        npix = point - (lobe - 10);
        if ( (npix >= 0) && (npix < n) )
            xp[point] += sinc[lobe] * x[npix];
        else
            xp[point] += sinc[lobe] * hole;
    }
}
Testbed code performance

![Graph showing compute time vs. image size for CPU performance.](image-url)
Which function should we port?

- `mpd_ipImageS_Shift2d_fs4 (proc_cpu)` vs `mpd_ipImageS_sshift_fnV5_VK (sshift)`

<table>
<thead>
<tr>
<th>% time</th>
<th>cumulative seconds</th>
<th>self seconds</th>
<th>calls</th>
<th>self s/call</th>
<th>total s/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>84.17</td>
<td>82.39</td>
<td>82.39</td>
<td>91504</td>
<td>0</td>
<td>0</td>
<td><code>mpd_ipImageS_sshift_fnV5_VK</code></td>
</tr>
<tr>
<td>7.49</td>
<td>89.72</td>
<td>7.33</td>
<td>152</td>
<td>0.05</td>
<td>0.59</td>
<td><code>mpd_ipImageS_Shift2d_fs4</code></td>
</tr>
<tr>
<td>5.39</td>
<td>95</td>
<td>5.28</td>
<td>152</td>
<td>0.03</td>
<td>0.63</td>
<td><code>mpd_fdel0d_fs7</code></td>
</tr>
<tr>
<td>0.87</td>
<td>95.85</td>
<td>0.85</td>
<td>65</td>
<td>0.01</td>
<td>0.01</td>
<td><code>mpd_LmmS_AlphaBetaSet_fs1</code></td>
</tr>
<tr>
<td>0.5</td>
<td>96.34</td>
<td>0.49</td>
<td>152</td>
<td>0</td>
<td>0</td>
<td><code>mpd_ipImageS_Add_star_fs7</code></td>
</tr>
<tr>
<td>0.3</td>
<td>96.63</td>
<td>0.29</td>
<td>142</td>
<td>0</td>
<td>0</td>
<td><code>mpd_fmodel_fs7</code></td>
</tr>
<tr>
<td>0.15</td>
<td>96.78</td>
<td>0.15</td>
<td></td>
<td></td>
<td></td>
<td><code>snorm</code></td>
</tr>
</tbody>
</table>
sshift CPU/MAP partitioning

- **CPU**
  - data management, as before

- **MAP**
  - 1D convolution

---

**CPU**

- Data management, as before.

**MAP**

- 1D convolution.
**sshift** - CPU code - I

```c
int main (int argc, char *argv[])  
{
   ...
   map_allocate(1);
   /* allocate images per command line arguments */
   ...
   gettimeofday(&time0, NULL);

   proc_cpu(image1, image2, dx, dy);

   gettimeofday(&time1, NULL);

   /* print time difference */
   ...
   map_free(1);
}

void sshift(float *x, long n, float shift, float hole,  
            float *xp, float *sinc)
{
   ...
   /* do the fractional shift (if any) */
   if ( fshift is significant )
   {
      /* convolve the input data with the sinc array */
      
      intp_shift ((int64_t *)sinc,  
                  (int64_t *)x,  
                  (int64_t)(n/2),  
                  (int64_t *)xp,  
                  hole,  
                  firsttime,  
                  &tm,  
                  mapnum);
   }
   ....
```
**sshift - CPU code - II**

```c
void init_sinc_array_MAP(float shift, float *sinc)
{
    int ishift;      /* Integer part of shift. */
    float fshift;    /* Fractional part of shift. */
    int point;       /* This is a counter variable. */
    float y;         /* Another counter variable. */

    ishift = (int)shift;
    fshift = shift - ishift;

    /* initialize the sinc array */
    for (y = fshift+10, point = 20 ; point >= 0; --y, point--)
        sinc[point] = (float)exp(-
            (y*y)/(DAMPFAC*DAMPFAC)) * sin(PI*y) /
            (PI*y);
}
```

```c
void init_sinc_array_CPU(float shift, float *sinc)
{
    int ishift;      /* Integer part of shift. */
    float fshift;    /* Fractional part of shift. */
    int point;       /* This is a counter variable. */
    float y;         /* Another counter variable. */

    ishift = (int)shift;
    fshift = shift - ishift;

    /* initialize the sinc array */
    for (y = fshift+10, point = 0 ; point < 21; --y, point++)
        sinc[point] = (float)exp(-
            (y*y)/(DAMPFAC*DAMPFAC)) * sin(PI*y) /
            (PI*y);
}
```
**ssshift - MAP code outline**

1. **Transfer image data to an OBM bank**
2. **Transfer coefficients to an OBM bank**
3. **Load coefficients from the OBM bank to on-chip registers**
4. **Read pixel value from the OBM bank**
5. **Compute convolution**
6. **Store results to the OBM bank**
7. **Transfer image data to the host**
```c
void intp_shift(int64_t sinc[], int64_t x[], int64_t n, int64_t xp[], float hole, int firsthand, int64_t *tm, int mapnum)
{
    // filter coefficients
    float ca00, ca01, ca02, ca03, ca04, ca05, ca06, ca07, ca08, ca09, ca10;
    float ca11, ca12, ca13, ca14, ca15, ca16, ca17, ca18, ca19, ca20, ca21;

    // pixels
    float p00, p01, p02, p03, p04, p05, p06, p07, p08, p09;
    float p10, p11, p12, p13, p14, p15, p16, p17, p18, p19, p20;

    // misc variables
    int64_t val, i, j, k;
    float v0, v1, v2, v3, v4, v5, vout;

    // input image
    OBM_BANK_A (AL, int64_t, MAX_OBM_SIZE)

    // output image
    OBM_BANK_B (BL, int64_t, MAX_OBM_SIZE)

    // filter coefficients
    OBM_BANK_F (FL, int64_t, MAX_OBM_SIZE)
```
sshift - MAP code - II

// raster line DMA data transfer
DMA_CPU (CM2OBM, AL, MAP_OBM_stripe(1,"A"), x, 1, n*sizeof(int64_t), 0);
wait_DMA(0);

if (firsttime == 0) // filter coefficients DMA data transfer
{
    DMA_CPU (CM2OBM, FL, MAP_OBM_stripe(1,"F"), sinc, 1, 11*sizeof(int64_t), 0);
    wait_DMA(0);
}

for (i = 0; i < 11; i++)
{
    ca00 = ca02;  ca01 = ca03;  ca02 = ca04;
    ca03 = ca05;  ca04 = ca06;  ca05 = ca07;
    ca06 = ca08;  ca07 = ca09;  ca08 = ca10;
    ca09 = ca11;  ca10 = ca12;  ca11 = ca13;
    ca12 = ca14;  ca13 = ca15;  ca14 = ca16;
    ca15 = ca17;  ca16 = ca18;  ca17 = ca19;
    ca18 = ca20;  ca19 = ca21;

    split_64to32_flt_flt (FL[i], &v1, &v0);
    ca20 = v0;  ca21 = v1;
}
for (i = 0; i < (2*n)+10; i++)
{
    cg_count ceil 64 (1, 0, i==0, 1, &k);
    cg_count ceil 64 (k==0, 0, i==0, INT_MAX, &j);

    if (k == 0)
        if (j < n)
            split 64to32 flt flt (AL[j], &v3, &v2);
        else
            { v2 = hole; v3 = hole; }
    else
        { p00 = p01; p01 = p02; p02 = p03; p03 = p04; p04 = p05; p05 = p06;
          p06 = p07; p07 = p08; p08 = p09; p09 = p10; p10 = p11; p11 = p12;
          p12 = p13; p13 = p14; p14 = p15; p15 = p16; p16 = p17; p17 = p18;
          p18 = p19; p19 = p20; }

    if (k == 0)
        p20 = v2;
    else
        p20 = v3;

    read pixel value from the OBM bank
if (j > 4)
    vout = ca00 * p00 + ca01 * p01 + ca02 * p02 + ca03 * p03 + ca04 * p04 + ca05 * p05 +
        ca06 * p06 + ca07 * p07 + ca08 * p08 + ca09 * p09 + ca10 * p10 + ca11 * p11 +
        ca12 * p12 + ca13 * p13 + ca14 * p14 + ca15 * p15 + ca16 * p16 + ca17 * p17 +
        ca18 * p18 + ca19 * p19 + ca20 * p20;

if (k == 0)
    v4 = vout;
else
{
    v5 = vout;

    if (j > 4)
    {
        comb_32to64_flt_flt (v5, v4, &val);
        BL[j-5] = val;
    }
}

DMA_CPU (OBM2CM, BL, MAP_OBM_stripe(1,"B"), xp, 1, n*sizeof(int64_t), 0);
wait_DMA (0);
\textit{sshift} - MAP code compilation

- Compiled code characteristics
  - \( \text{freq} = 96.7 \text{ MHz} \)
  - device utilization summary:
    - Number of \text{MULT18X18s}
      - 84 out of 144 \( 58\% \)
    - Number of \text{RAMB16s}
      - 0 out of 144 \( 0\% \)
    - Number of \text{SLICEs}
      - 23519 out of 33792 \( 69\% \)
- overall compilation time
  - 2 hours 56 minutes (P&R really)
Code performance - I

![Code performance comparison chart]

- **speed-up factor**
- **CPU**
- **MAP**

### Compute time (s)

<table>
<thead>
<tr>
<th>Image Size (pixels)</th>
<th>Compute Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>0.1</td>
</tr>
<tr>
<td>256</td>
<td>0.2</td>
</tr>
<tr>
<td>512</td>
<td>0.5</td>
</tr>
<tr>
<td>1024</td>
<td>0.9</td>
</tr>
<tr>
<td>2048</td>
<td>1.5</td>
</tr>
<tr>
<td>4096</td>
<td>2.1</td>
</tr>
<tr>
<td>8192</td>
<td>2.3</td>
</tr>
</tbody>
</table>
Code performance - II

The diagram shows a comparison of compute time for different image sizes, with two types of performance: I/O only and I/O+compute.

- For small image sizes (128-512 pixels), the compute time is minimal and does not significantly differ between I/O only and I/O+compute.
- As the image size increases to 1024 pixels, the compute time begins to rise, especially for I/O+compute.
- A significant increase in compute time is observed for image sizes of 2048 and 4096 pixels, with I/O+compute becoming the dominant factor.

The graph highlights the importance of efficient I/O handling as image sizes grow, to maintain performance levels that are competitive with pure compute tasks.
Code performance

• Why is that?

<table>
<thead>
<tr>
<th>%</th>
<th>cumulative</th>
<th>self</th>
<th>calls</th>
<th>self</th>
<th>total</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time</td>
<td></td>
<td>seconds</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>seconds</td>
<td></td>
<td>seconds</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>84.17</td>
<td>82.39</td>
<td>82.39</td>
<td>91504</td>
<td>0</td>
<td>0</td>
<td>mpd_ipImageS_sshift_fnV5_VK</td>
</tr>
<tr>
<td>7.49</td>
<td>89.72</td>
<td>7.33</td>
<td>152</td>
<td>0.05</td>
<td>0.59</td>
<td>mpd_ipImageS_Shift2d_fs4</td>
</tr>
<tr>
<td>5.39</td>
<td>95</td>
<td>5.28</td>
<td>152</td>
<td>0.03</td>
<td>0.63</td>
<td>mpd_fdeIs0d_fs7</td>
</tr>
<tr>
<td>0.87</td>
<td>95.85</td>
<td>0.85</td>
<td>65</td>
<td>0.01</td>
<td>0.01</td>
<td>mpd_LmmS_AlphaBetaSet_fs1</td>
</tr>
<tr>
<td>0.5</td>
<td>96.34</td>
<td>0.49</td>
<td>152</td>
<td>0</td>
<td>0</td>
<td>mpd_ipImageS_Add_star_fs7</td>
</tr>
<tr>
<td>0.3</td>
<td>96.63</td>
<td>0.29</td>
<td>142</td>
<td>0</td>
<td>0</td>
<td>mpd_fmodel_fs7</td>
</tr>
<tr>
<td>0.15</td>
<td>96.78</td>
<td>0.15</td>
<td></td>
<td></td>
<td></td>
<td>snorm</td>
</tr>
</tbody>
</table>

• So, which function should we port?
  – mpd_ipImageS_Shift2d_fs4 (proc_cpu) OR
  – mpd_ipImageS_sshift_fnV5_VK (ssshift)
**proc_fpga** CPU/MAP partitioning

- **CPU**
  - data management, as before

- **MAP**
  - 2D convolution

![Diagram showing CPU and MAP partitioning]

- **FPGA 1**
  - OBM A
  - OBM B
  - OBM C

- **FPGA 2**
  - OBM D
  - OBM E
  - OBM F
**proc_fpga - CPU code - I**

```c
int main (int argc, char *argv[]) {
    .
    .
    map_allocate(1);
    .
    /* allocate images per command line arguments */
    .
    gettimeofday(&time0, NULL);
    proc_fpga(image1, image2, dx, dy);
    gettimeofday(&time1, NULL);
    /* print time difference */
    .
    .
    map_free(1);
    .
}
```

```c
int proc_fpga(pgm_image *image_in,
              pgm_image *image_out, float dx, float dy)
{
    .
    .
    /* compute sinc functions */
    init_sinc_array_MAP(dx, sinc_x);
    init_sinc_array_MAP(dy, sinc_y);
    intp_filter ((int64_t *)image_in->img,
                 (int64_t *)image_out->img,
                 (int)image_in->sm,
                 (int)image_in->sn,
                 (int64_t *)sinc_x,
                 (int64_t *)sinc_y,
                 zeroF, image_in->sm*image_in->sn*sizeof(int64_t)/2,
                 &tm, mapnum);
    .
    .
}
```
**proc_fpga** - MAP details - I

**primary chip**

- transfer 2 sets of coefficients to separate OBM banks
- load one set of coefficients from the OBM bank to on-chip registers
- transfer image data to 3 OBM banks
- compute convolution for each row, one pixel at a time
- transfer image data out of 3 OBM banks to the host

**secondary chip**

- OBM A-C
- OBM D-F
- OBM E, F
- load one set of coefficients from the OBM bank to on-chip registers
- compute convolution for each column, one pixel at a time
**proc_fpga - MAP details- II**

- **Image size is limited to 12 MB**
  - 3 OBM banks are used to store input image
  - 3 OBM banks are used to store output image

- **Only one pixel is computed at each loop cycle**
  - Not enough space on the chip (V6000) to accommodate 2 pixel calculations per single pass of the loop
**proc_fpga - MAP code compilation**

**Primary chip**
- freq = 95.6 MHz
- device utilization summary:
  - Number of MULT18X18s
    - 93 out of 144  64%
  - Number of RAMB16s
    - 0 out of 144  0%
  - Number of SLICEs
    - 26702 out of 33792  79%
- Overall compilation time
  - >3 hours

**Secondary chip**
- freq = 85.4 MHz
- device utilization summary:
  - Number of MULT18X18s
    - 96 out of 144  66%
  - Number of RAMB16s
    - 4 out of 144  2%
  - Number of SLICEs
    - 24582 out of 33792  72%
- Overall compilation time
  - <4 hours

National Center for Supercomputing Applications
Code performance - I

- **Speed-up factor**
- **CPU**
- **MAP**

The graph compares the compute time (in seconds) for different image sizes (in pixels). The speed-up factor is shown, with CPU performance indicated by a solid line and MAP performance by a dashed line. The x-axis represents the image size, and the y-axis represents the compute time. Notable speed-up factors are indicated, such as ~39 and ~105.
Code performance - II

[Graph showing compute vs I/O performance comparison chart]

- I/O only
- I/O+compute

Image size (pixels):
- 128
- 256
- 512
- 1024
- 1772
- 2048
- 4096

Compute time (s)
Code performance - III

- **1st implementation**
  - multiple calls to MAP

- **2nd implementation**
  - single call to MAP
What to do about smaller images?

- Q: How can we speedup calculations for images of a smaller size?
- A: pack them together so that there is a fewer MAP function calls overall!

\[
\begin{array}{cc|c}
512x512 & 512x512 & 512x512 \\
512x512 & 512x512 & 512x512 \\
\hline
1024x1024 & 1024x1024 & 1024x1024 \\
\end{array}
\]

= 4 MB = 1 OBM bank

but since there are 3 OBM banks, we can load 12 512x512 images at once
## Code performance - IV

<table>
<thead>
<tr>
<th>Image size</th>
<th>Number of images</th>
<th>CPU sec/image</th>
<th>MAP seconds</th>
<th>MAP sec/image</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1772x1772</td>
<td>1 (≈12 MB)</td>
<td>2.01</td>
<td>0.24</td>
<td>0.24</td>
<td>x8.5</td>
</tr>
<tr>
<td>1024x1024</td>
<td>3 (12 MB)</td>
<td>0.86</td>
<td>0.24ₜ</td>
<td>0.24/3≈0.08</td>
<td>~x8.5</td>
</tr>
<tr>
<td>512x512</td>
<td>12 (12 MB)</td>
<td>0.20</td>
<td>0.24ₜ</td>
<td>0.24/12≈0.02</td>
<td>~x8.5</td>
</tr>
<tr>
<td>256x256</td>
<td>48 (12 MB)</td>
<td>0.04</td>
<td>0.24ₜ</td>
<td>0.24/48≈0.005</td>
<td>~x8.5</td>
</tr>
<tr>
<td>128x128</td>
<td>192 (12 MB)</td>
<td>0.01</td>
<td>0.24ₜ</td>
<td>0.24/192≈0.00125</td>
<td>~x8.5</td>
</tr>
</tbody>
</table>
Next generation MAP - I

• Calculate two pixels per pass on VP100
  – Gives a ~x2 speedup

• Modify the code to handle four pixels per pass
  – Calculate two pixels on each FPGA
  – Gives another ~x2 speedup

• Modify the code to perform streaming of input and output images
  – This eliminates the latency of input and output DMAs
  – Image size is doubled (24MB)

• VP100 are faster
  – 135 MHz compared to 100 MHz for V6000
Next generation MAP - II

primary chip

1. Transfer 2 sets of coefficients to OBM banks
2. Load both sets of coefficients
3. Stream image data
4. Compute convolution for each row, two pixels at a time
5. Compute convolution for each column, two pixels at a time
6. Stream image data out

secondary chip

1. Load both set of coefficients
2. Stream image data
3. Compute convolution for each row, two pixels at a time
4. Compute convolution for each column, two pixels at a time
Lessons learned

• **What routines to port to MAP?**
  – CPU intensive
    • `mpd_iplImageS_sshift_fnV5_VK`
    • `mpd_iplImageS_Shift2d_fs4`
  – do a lot of compute per input element(s)
    • `mpd_iplImageS_sshift_fnV5_VK`
    • `mpd_iplImageS_Shift2d_fs4`
  – maximize compute to I/O ratio
    • `mpd_iplImageS_sshift_fnV5_VK`
    • `mpd_iplImageS_Shift2d_fs4`
Lessons learned

• **MAP routine optimization**
  – Utilize computational parallelism
    • loop unrolling
    • parallel sections
    • two chips
  – Utilize streams
    • to maximize DMA and computational concurrency
      – minimizes the latency effect of DMAs by overlapping compute with DMAs
    • to overlap computational loops that are serial in nature
      – minimizes the latency effect of serial computations
Conclusions

• Speed-up is ~8.5 times as compared to the same code running on CPU
  – More is expected on the next generation MAP

• Fairly simple and straightforward implementation of the code on MAP
  – Brining data in/out and dealing with OBM banks is the only “unfamiliar” concept to a regular C programmer
Acknowledgements

• Kenneth Mighell, National Optical Astronomy Observatory
• David Caliga, SRC Computers, Inc.