Improving High Performance Image Resizing and Rotation: A Case Study of Texture Options

Ismayil Güracar
Senior Key Expert
Siemens Medical Solutions USA, Inc
Ultrasound Business Unit

GTC2016 S6197 Wednesday April 6, 3:00pm LL21B
The Problem: resize and rotate 2D images

Interpolation Methods: Bilinear and Cubic (Cattmull-Rom)

Memory Storage and Access:
Texture based: CUDA Array, linear memory using Tex1Dfetch, Pitched 2D array
Memory array based: linear memory using device pointers

Data Types: float, float2, float4
i.e. float \rightarrow \text{greyscale image}, \text{float2} \rightarrow \text{complex data}, \text{float4} \rightarrow \text{RGBA or other vector data type}

Measurements: output data rate performance, Nsight profiler memory statistics
### Device memory access methods

<table>
<thead>
<tr>
<th></th>
<th>Kernel Read API</th>
<th>Built-in Bilinear Interpolation available?</th>
<th>Kernel Write API (caveats(^1))</th>
<th>Host read/write API</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CUDA Array</strong></td>
<td>tex2D</td>
<td>Yes, With 8-bit precision coefficients</td>
<td>surf2Dwrite(^1)</td>
<td>cudaMemcpyArray</td>
</tr>
<tr>
<td><strong>Pitched 2D Array</strong></td>
<td>tex2D or flat memory access d_ptr[index] *d_ptr</td>
<td>Yes, With 8-bit precision coefficients</td>
<td>flat memory access d_ptr[index] *d_ptr</td>
<td>cudaMemcpy</td>
</tr>
<tr>
<td><strong>Linear Memory</strong></td>
<td>tex1Dfetch or flat memory access d_ptr[index] *d_ptr</td>
<td>No</td>
<td>flat memory access d_ptr[index] *d_ptr</td>
<td>cudaMemcpy</td>
</tr>
</tbody>
</table>

\(^1\) Texture cache coherency within kernel execution not guaranteed—so don’t write to a texture and then read it in the same kernel launch.
Bilinear interpolation

linear combination of 4 nearest neighbors with weights based on fractional distance $\alpha$ and $\beta$

$$I_0 = V_{00} + \alpha (V_{01} - V_{00})$$
$$I_1 = V_{10} + \alpha (V_{11} - V_{10})$$

Result = $I_0 + \beta (I_1 - I_0)$

Texture unit has built-in hardware for doing bilinear interpolation calculation

```cpp
if (INTERPOLATION_TYPE_SELECT == Linear)
{
    V = tex2D<DATA_TYPE>(tex_inArg, fX+0.5f, fY+0.5f);
}
```
Cubic interpolation

- Linear combination of 16 nearest neighbors
- Separable operation—first interpolate in x and then in y or vice versa
- Currently **No** built in hardware on the GPU for cubic interpolation

Sample read using either texture lookups or explicit device memory access using pointers
Read in 16 neighbor points, using indices that are quantized to avoid bilinear interpolation by the texture unit or disabling the interpolation i.e. `filterMode = cudaFilterModePoint`

```
DATA_TYPE V00, V10, V20, V30, V01, V11, V21, V31, V02, V12, V22, V32, V03, V13, V23, V33;
V00 = tex2D<DATA_TYPE>(tex_inArg, ffX+0.0f, ffY+0.0f);
V10 = tex2D<DATA_TYPE>(tex_inArg, ffX+1.0f, ffY+0.0f);
V20 = tex2D<DATA_TYPE>(tex_inArg, ffX+2.0f, ffY+0.0f);
V30 = tex2D<DATA_TYPE>(tex_inArg, ffX+3.0f, ffY+0.0f);
V01 = tex2D<DATA_TYPE>(tex_inArg, ffX+0.0f, ffY+1.0f);
V11 = tex2D<DATA_TYPE>(tex_inArg, ffX+1.0f, ffY+1.0f);
V21 = tex2D<DATA_TYPE>(tex_inArg, ffX+2.0f, ffY+1.0f);
V31 = tex2D<DATA_TYPE>(tex_inArg, ffX+3.0f, ffY+1.0f);
V02 = tex2D<DATA_TYPE>(tex_inArg, ffX+0.0f, ffY+2.0f);
V12 = tex2D<DATA_TYPE>(tex_inArg, ffX+1.0f, ffY+2.0f);
V22 = tex2D<DATA_TYPE>(tex_inArg, ffX+2.0f, ffY+2.0f);
V32 = tex2D<DATA_TYPE>(tex_inArg, ffX+3.0f, ffY+2.0f);
V03 = tex2D<DATA_TYPE>(tex_inArg, ffX+0.0f, ffY+3.0f);
V13 = tex2D<DATA_TYPE>(tex_inArg, ffX+1.0f, ffY+3.0f);
V23 = tex2D<DATA_TYPE>(tex_inArg, ffX+2.0f, ffY+3.0f);
V33 = tex2D<DATA_TYPE>(tex_inArg, ffX+3.0f, ffY+3.0f);
```
if (INTERPOLATION_TYPE_SELECT == CubicCatmullRom) {
    float ffX = floor(fX-2.0f)+0.5f;
    float ffY = floor(fY-2.0f)+0.5f;
    float c0,c1,c2,c3,d0,d1,d2,d3;
    getCubicInterpCoefs(fX, &c0, &c1, &c2, &c3);
    getCubicInterpCoefs(fY, &d0, &d1, &d2, &d3);
}

__device__ __inline__ void getCubicInterpCoefs(float coord, float* c0, float* c1, float* c2, float* c3 ) {
    float frac,frac2,frac3;
    frac = coord - floorf(coord);
    frac2=frac*frac;
    frac3=frac*frac2;

    *c0 = Coef03*frac3 + Coef02*frac2 + Coef01*frac + Coef00;
    *c1 = Coef13*frac3 + Coef12*frac2 + Coef11*frac + Coef10;
    *c2 = Coef23*frac3 + Coef22*frac2 + Coef21*frac + Coef20;
    *c3 = Coef33*frac3 + Coef32*frac2 + Coef31*frac + Coef30;
}

Evaluate cubic polynomial to compute filter coefficients
Get coefficients for separable filter in x and y
Finally, compute the dot product of coefficients and neighborhood data points

Result = c0*d0*V00 + c0*d1*V01 + c0*d2*V02 + c0*d3*V03 
+ c1*d0*V10 + c1*d1*V11 + c1*d2*V12 + c1*d3*V13 
+ c2*d0*V20 + c2*d1*V21 + c2*d2*V22 + c2*d3*V23 
+ c3*d0*V30 + c3*d1*V31 + c3*d2*V32 + c3*d3*V33;
Tradeoff—computation and memory access cost versus image quality

Bilinear interpolation (zoom factor = 8)
  Single texture lookup per pixel out

Cubic Catmull-Rom Interpolation (zoom factor=8)
  16-lookups per pixel out
Experiments:

Resize and rotate images using various data access methods for bilinear and cubic interpolation filtering, using native texture bilinear interpolation when possible.

Measure performance for varying conditions and use Nsight to gain understanding

- `tex2D` from `cudaArray`

- `tex2D cudaArray + copy from device memory to cudaArray`
  Copy required if data not already in a `cudaArray`

- `tex2D from pitch2D array`
  pitch2D array often easier to use with kernels and libraries, and avoids copy to `cudaArray`

- `tex1Dfetch with explicit coefficient computation`

- Flat memory array with explicit coefficient computation
Testbed—a simple kernel. Code provided in supplementary slide.

One thread per output pixel—lookup neighborhood samples with read addressing based on sin/cos of rotation angle and zoom factor

First set of experiments—don’t resize or rotate image and see what happens with each access method while varying the width of the image data—float, float2, or float4
### BILINEAR INTERPOLATION

<table>
<thead>
<tr>
<th>Storage type</th>
<th>Data type</th>
<th>Processing rate Msamples/sec</th>
<th>Relative perf Scaled by width</th>
</tr>
</thead>
<tbody>
<tr>
<td>tex2D from cudaArray</td>
<td>float1</td>
<td>7821</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>float2</td>
<td>3890</td>
<td>99.5%</td>
</tr>
<tr>
<td></td>
<td>float4</td>
<td>1874</td>
<td>95.9%</td>
</tr>
<tr>
<td>tex2D cudaArray + copy to cudaArray</td>
<td>float1</td>
<td></td>
<td>49.7%</td>
</tr>
<tr>
<td></td>
<td>float2</td>
<td></td>
<td>49.2%</td>
</tr>
<tr>
<td></td>
<td>float4</td>
<td></td>
<td>48.9%</td>
</tr>
<tr>
<td>tex2D from pitch2D array</td>
<td>float1</td>
<td></td>
<td>100.5%</td>
</tr>
<tr>
<td></td>
<td>float2</td>
<td></td>
<td>100.7%</td>
</tr>
<tr>
<td></td>
<td>float4</td>
<td></td>
<td>101.6%</td>
</tr>
<tr>
<td>tex1Dfetch with Explicit coefficient computation</td>
<td>float1</td>
<td></td>
<td>82.6%</td>
</tr>
<tr>
<td></td>
<td>float2</td>
<td></td>
<td>100.9%</td>
</tr>
<tr>
<td></td>
<td>float4</td>
<td></td>
<td>101.1%</td>
</tr>
<tr>
<td>Flat memory array with explicit coefficient computation</td>
<td>float1</td>
<td></td>
<td>81.4%</td>
</tr>
<tr>
<td></td>
<td>float2</td>
<td></td>
<td>100.8%</td>
</tr>
<tr>
<td></td>
<td>float4</td>
<td></td>
<td>100.8%</td>
</tr>
</tbody>
</table>

**zoom x 1**

- 2kx2k in, 2kx2k out
- No rotation
- K2200 Maxwell

- Memory copy to cudaArray
  - Halves performance—performance
  - Dominated by device memory BW

- Cost of explicit coefficient computation is exposed
## CUBIC INTERPOLATION

<table>
<thead>
<tr>
<th>Storage type</th>
<th>Data type</th>
<th>Processing rate Msamples/sec</th>
<th>Relative perf Scaled by width</th>
</tr>
</thead>
<tbody>
<tr>
<td>tex2D from cudaArray explicit coefficient computation</td>
<td>float1</td>
<td>2391</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>float2</td>
<td>1226</td>
<td>102%</td>
</tr>
<tr>
<td></td>
<td>float4</td>
<td>515</td>
<td>86%</td>
</tr>
<tr>
<td>tex2D cudaArray + copy to cudaArray</td>
<td>float1</td>
<td></td>
<td>80%</td>
</tr>
<tr>
<td></td>
<td>float2</td>
<td></td>
<td>81%</td>
</tr>
<tr>
<td></td>
<td>float4</td>
<td></td>
<td>68%</td>
</tr>
<tr>
<td>tex2D from pitch2D array with explicit coefficient computation</td>
<td>float1</td>
<td></td>
<td>108%</td>
</tr>
<tr>
<td></td>
<td>float2</td>
<td></td>
<td>118%</td>
</tr>
<tr>
<td></td>
<td>float4</td>
<td></td>
<td>99%</td>
</tr>
<tr>
<td>tex1Dfetch with explicit coefficient computation</td>
<td>float1</td>
<td></td>
<td>102%</td>
</tr>
<tr>
<td></td>
<td>float2</td>
<td></td>
<td>192%</td>
</tr>
<tr>
<td></td>
<td>float4</td>
<td></td>
<td>195%</td>
</tr>
<tr>
<td>Flat memory array with explicit coefficient computation</td>
<td>float1</td>
<td></td>
<td>121%</td>
</tr>
<tr>
<td></td>
<td>float2</td>
<td></td>
<td>129%</td>
</tr>
<tr>
<td></td>
<td>float4</td>
<td></td>
<td>138%</td>
</tr>
</tbody>
</table>

zoom x 1
2kx2k in , 2kx2k out
No rotation
K2200 Maxwell

- Reduced occupancy due to register pressure
- Memory copy to cudaArray 20% hit to performance which is dominated by cubic filter computation and large number of texture reads
- Interesting!—tex1Dfetch does a lot better in some cases With wider data...Let's look with Nsight later
Second set of experiments – vary zoom factor and see how performance varies
Performance Variation with Zoom Factor

Bilinear Interpolation float1

Memory Copy is a large fraction of the cost for cudaArray with copy.
Performance Variation with Zoom Factor

Bilinear Interpolation float1

Caution: quantization in built-in interpolation coefficients may be trouble at high zoom factors!
Third set of experiments – vary rotation angle factor and see how performance varies
Performance Variation with Rotation Angle

Bilinear Interpolation float1

Graph showing performance variation with rotation angle for different methods.

- cudaArray
- cudaArray with copy
- pitch2D
- tex1Dfetch
- flat memory
Performance Variation with Rotation Angle

Bilinear Interpolation float1

Nsight Memory Statistics Overview

<table>
<thead>
<tr>
<th>Name</th>
<th>Total</th>
<th>Per Warp</th>
<th>Per Second</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads - SM from L1/Tex Cache</td>
<td>131,072.00</td>
<td>1.00</td>
<td>247,896,900.00</td>
</tr>
<tr>
<td>Fetches</td>
<td>131,072.00</td>
<td>1.00</td>
<td>247,896,900.00</td>
</tr>
<tr>
<td>L1/Tex Transactions</td>
<td>2,097,152.00</td>
<td>16.00</td>
<td>3,966,350,000.00</td>
</tr>
<tr>
<td>Size</td>
<td>16.00 MB</td>
<td>128.00 B</td>
<td>29.55 GB/s</td>
</tr>
<tr>
<td>Cache Bank Conflicts</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Cache Hit Rate</td>
<td>1.98 %</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Performance Variation with Rotation Angle

Bilinear Interpolation float1

Performance drops due to 2x higher number of texture transactions, high rate of cache bank conflicts and low cache hit rate.
Cache Hit Rate with Varying Rotation Angle and Tiling

Input Pixels sampled by a single warp with bilinear interpolation to produce output pixels
Green dots – samples that are actually used to produce an output value
Red dots – samples that are loaded into cache, but not used

0 degree rotation angle: good cache hit rate

45 degree rotation angle: poor cache hit ratio

Y-Tiling (by 8) Improves cache hit ratio for large rotation angles

Cache line size = 32 bytes with float data, 8 pixels
Bilinear Interpolation float1

Performance variation with rotation angle

With tiling

- cudaArray
- cudaArray with copy
- pitch2D
- tex1Dfetch
- flat memory

Tile Size 128 × 1

Tile Size 64 × 8
Bilinear Interpolation float1

Performance variation with rotation angle

Tiling improves cache hit rate

Nsight Memory Statistics Overview

<table>
<thead>
<tr>
<th>Name</th>
<th>Total</th>
<th>Per Warp</th>
<th>Per Second</th>
</tr>
</thead>
<tbody>
<tr>
<td>With tiling</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fetches</td>
<td>131,072.00</td>
<td>1.00</td>
<td>259,898,500.00</td>
</tr>
<tr>
<td>L2 Transactions</td>
<td>4,739,187.00</td>
<td>36.16</td>
<td>9,397,181,000.00</td>
</tr>
<tr>
<td>Size</td>
<td>16.00 MB</td>
<td>128.00 B</td>
<td>30.98 GB/s</td>
</tr>
<tr>
<td>Cache Bank Conflicts</td>
<td>56,916.00</td>
<td>0.43</td>
<td>112,856,900.00</td>
</tr>
<tr>
<td>Cache Hit Rate</td>
<td>50.16 %</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Total</th>
<th>Per Warp</th>
<th>Per Second</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without tiling</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fetches</td>
<td>131,072.00</td>
<td>1.00</td>
<td>200,322,800.00</td>
</tr>
<tr>
<td>L1/Tex Transactions</td>
<td>4,736,744.00</td>
<td>36.15</td>
<td>7,242,420,000.00</td>
</tr>
<tr>
<td>Size</td>
<td>16.00 MB</td>
<td>128.00 B</td>
<td>23.88 GB/s</td>
</tr>
<tr>
<td>Cache Bank Conflicts</td>
<td>57,057.00</td>
<td>0.44</td>
<td>87,202,580.00</td>
</tr>
<tr>
<td>Cache Hit Rate</td>
<td>27.97 %</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Cubic Interpolation float1

Performance variation with rotation angle

Perf relative to Cubic Interpolation float1 cudaArray with 0 degree rotation

With tiling, pitch2D texture performs similar to cudaArray texture lookup

pitch2D outperforms cudaArray with copy!
With tiling, pitch2D texture performs similarly to cudaArray texture lookup.

Without tiling at 45 degrees rotation, pitch2D texture has a lower cache hit rate leading to a big increase in number of L2 cache loads.
Cubic Interpolation float2 Performance variation with rotation angle

Perf relative to Cubic Interpolation float1 cudaArray with 0 degree rotation

With float2, pitch2D textures have a greater number of texture transactions compared to cudaArray texture lookups

→ Advantage of special data layout in cudaArrays
Cubic Interpolation float4

Perf relative to Cubic Interpolation float1 cudaArray with 0 degree rotation

Performance variation with rotation angle

- cudaArray
- cudaArray with copy
- pitch2D
- tex1Dfetch
- flat memory

With tiling

- cudaArray
- cudaArray with copy
- pitch2D
- tex1Dfetch
- flat memory

With float4

pitch2D performance degrades rapidly with increasing rotation angle

tex1D fetch and flat memory outperform other methods for small rotation angles
Cubic Interpolation float4

Perf relative to Cubic Interpolation float1 cudaArray with 0 degree rotation

Tex1Dfetch is more efficient—only lookup 1 value whereas tex2D looks up 4, even though we only need one of the values
Cubic Interpolation float4

At 45 degrees rotation Tex1Dfetch suffers from lower cache hit rate compared to cudaArray.

Performance variation with rotation angle

<table>
<thead>
<tr>
<th>Name</th>
<th>Total</th>
<th>Per Warp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads - SM from L1/Tex Cache</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fetches</td>
<td>2,097,152.00</td>
<td>16.00</td>
</tr>
<tr>
<td>L1/Tex Transactions</td>
<td>120,377,500.00</td>
<td>918.41</td>
</tr>
<tr>
<td>Size</td>
<td>256.00 MB</td>
<td>2.00 kB</td>
</tr>
<tr>
<td>Cache Bank Conflicts</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Cache Hit Rate</td>
<td></td>
<td>77.71 %</td>
</tr>
<tr>
<td>Loads - SM from L1/Tex Cache</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fetches</td>
<td>2,097,152.00</td>
<td>16.00</td>
</tr>
<tr>
<td>L1/Tex Transactions</td>
<td>56,188,180.00</td>
<td>428.68</td>
</tr>
<tr>
<td>Size</td>
<td>256.00 MB</td>
<td>2.00 kB</td>
</tr>
<tr>
<td>Cache Bank Conflicts</td>
<td>2,013.00</td>
<td>0.02</td>
</tr>
<tr>
<td>Cache Hit Rate</td>
<td>30.67 %</td>
<td></td>
</tr>
</tbody>
</table>
Cubic Interpolation float4

Perf relative to Cubic Interpolation float1 cudaArray with 0 degree rotation

The flat memory access has a very low cache hit rate and a high rate of cache bank conflicts.
Some Conclusions for Using Textures for Image Resizing and Rotation

• Memory access pattern is critical to performance—there is no single best method

• cudaArray with texture lookup offers significant benefits
  ➢ GPU is optimized for access patterns particularly seen in image rotation and resizing
  ➢ You have to pay the price to put your data into a cudaArray (cudaMemcpyToArray) or (surfWrite1D,2D) from a kernel

• pitched2D arrays allow texture filter hardware to be exploited without paying the conversion to cudaArray price—the extra memory copy or more effort in upstream kernels (surface writes)

• pitched2D arrays can perform nearly as well as cudaArray, if you take the extra effort to optimize access patterns (i.e. tiling)
Some More Conclusions for Using Textures for Image Resizing and Rotation

- `tex1Dfetch` is a good way to make use of the texture cache while avoiding unnecessary data access pressure required to feed `tex2D` interpolation hardware.
  - 1D fetch reads one sample, `tex2D` reads four samples per call

- Computational benefits of built-in bilinear interpolation texture hardware can be overshadowed by memory access and caching considerations—are you I/O bound or compute bound?

- Tiling can improve cache hit rate and overall performance
Thank You for Your Attention and Questions!

Ismayil Guracar
Senior Key Expert
Siemens Medical Solutions, USA Inc.
Ultrasound Business Unit

Mountain View, California
ismayil.guracar@siemens.com
template <int INTERPOLATION_TYPE_SELECT, typename DATA_TYPE >
__global__ void remapCombined_kernel(cudaTextureObject_t tex_inArg,
DATA_TYPE *d_out,
int xSizeIn, int ySizeIn, int xSizeOut, int ySizeOut,
int pitchInDATA_TYPE, int pitchOutDATA_TYPE,
float yPreScale, float xPreScale, float yPreOffset, float xPreOffset,
float cosTheta, float sinTheta )
{// each thread processes one output sample
int iFx = blockIdx.x*blockDim.x + threadIdx.x;
int iFy = blockIdx.y*blockDim.y + threadIdx.y;

int indexOut = iFy*pitchOutDATA_TYPE + iFx;
float Fx,Fy; DATA_TYPE V;
if((xFx<xSizeOut)&&(iFy<ySizeOut))
{
Fx = (float(iFx) + xPreOffset)*xPreScale + 0.5f;
Fy = (float(iFy) + yPreOffset)*yPreScale + 0.5f;
float ffX = Fx - xSizeIn/2;
float ffY = Fy - ySizeIn/2;
Fx= cosTheta*ffX + sinTheta*ffY + xSizeIn/2;
Fy=-sinTheta*ffX + cosTheta*ffY + ySizeIn/2;

int xCoordInt=(int)Fx;
int yCoordInt=(int)Fy;
int indexIn = xCoordInt + pitchInDATA_TYPE*(yCoordInt);
if (INTERPOLATION_TYPE_SELECT == Linear)
{
    // bilinear interpolation case
    DATA_TYPE V00,V01,V10,V11;
    V00 = tex1Dfetch<DATA_TYPE>(tex_inArg, indexIn);
    V10 = tex1Dfetch<DATA_TYPE>(tex_inArg, indexIn+1);
    V01 = tex1Dfetch<DATA_TYPE>(tex_inArg, indexIn+pitchInDATA_TYPE);
    V11 = tex1Dfetch<DATA_TYPE>(tex_inArg, indexIn+pitchInDATA_TYPE+1);
    float c0,c1,d0,d1;
    getLinearInterpCoefs( Fx, &c0, &c1 );
    getLinearInterpCoefs( Fy, &d0, &d1 );
    V=c0*d0*V00+c0*d1*V01+c1*d0*V10+c1*d1*V11;
}
}