Parallel Image/Video Editor

Adam Feinstein and H. Parker Shelton, 520.443, Johns Hopkins University

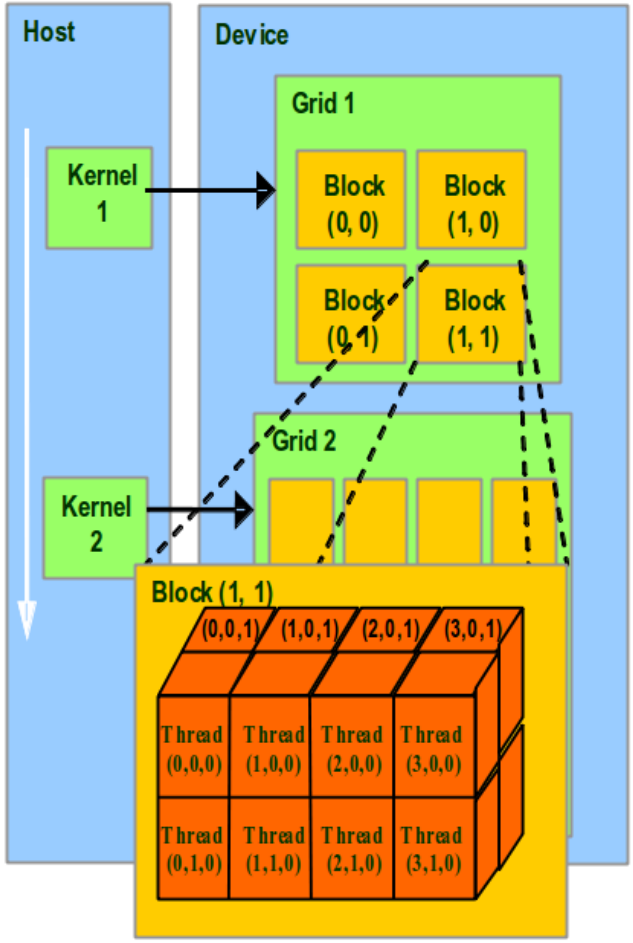
Project Overview

GPU-Parallelized Image and Video Editor

Multiple Encoding Algorithms

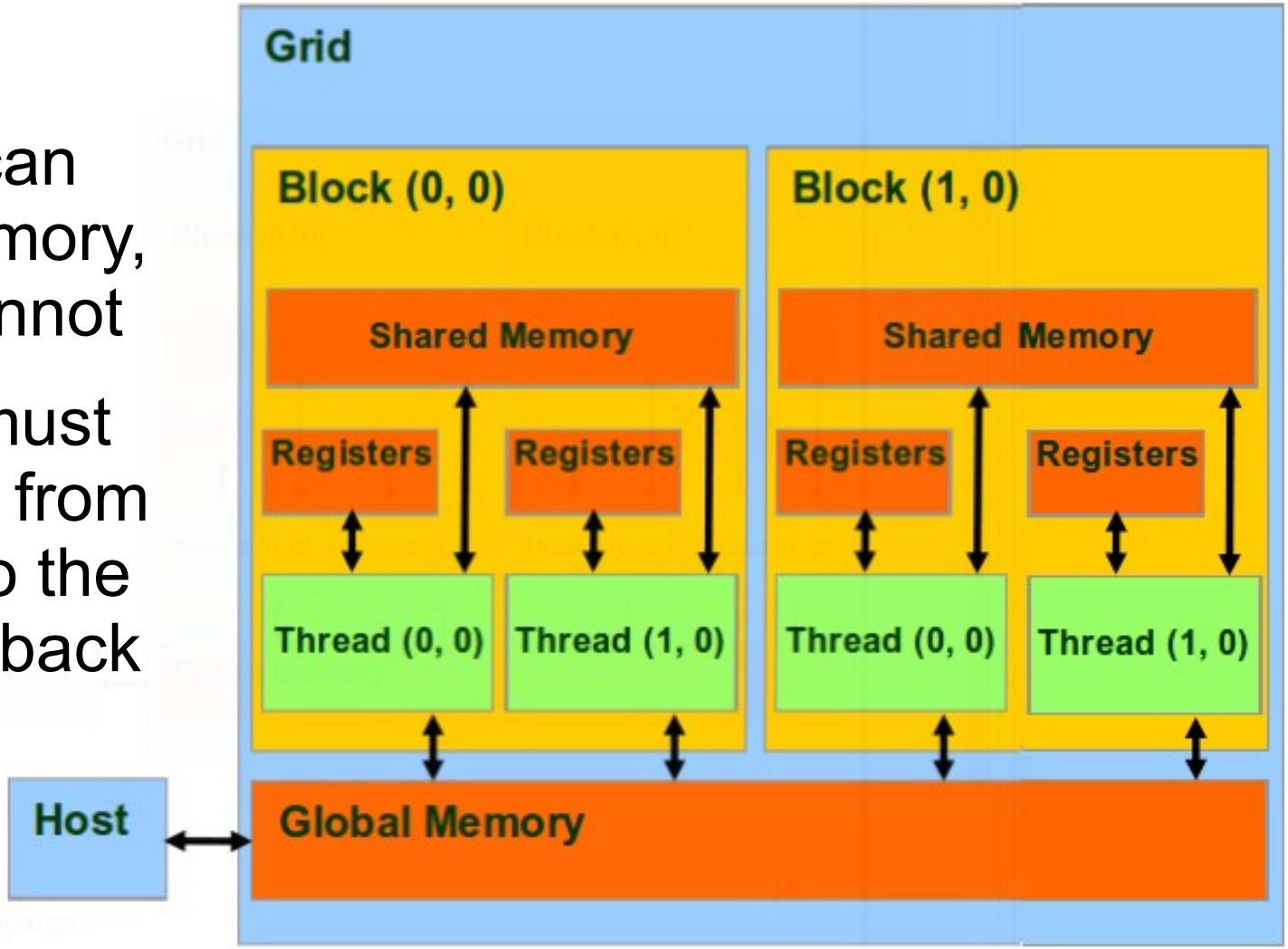
Nvidia's CUDA

- Compute-Unified Device Architecture
- GPU (grid) divided into Blocks, Threads
- Each Thread invokes the same kernel, preforms its own calculations



Nvidia's CUDA

- Threads can share memory, Blocks cannot
- Memory must be copied from the host to the GPU and back again



Qt

Cross-Platform C++ GUI Framework

Multi-threading, Multimedia, WebKit, SQL

GUI Demonstration

Image Editing

Grayscale

Brightness

Contrast

Saturation

Crop

Rotate

Scale

Blur

Edge Detect

Grayscale

```
// Replace each RGB with the luminance value


for(int y = 0; y < img->height(); y++) {
     for(int x = 0; x < img->width(); x++) {
          QRgb p = img->pixel(x, y);
          r = g = b = (qRed(p) * 0.3) + (qGreen(p) * 0.59) + (qBlue(p) * 0.11);
          img->setPixel(x, y, qRgb(r, g, b));
     }
}
return img;
```

GPU: Grayscale

```
extern "C" void CUgreyscale(unsigned char* output, unsigned char* input, int row, int col);

void CUgreyscale(unsigned char* output, unsigned char* input, int row, int col)
{
     dim3 dimGrid(row/16+1, col/16+1);
     dim3 dimThreadBlock(16,16);
     greyscale<<<dimGrid, dimThreadBlock>>>(input, output, row, col);
}

__global__ void greyscale(unsigned char* input, unsigned char* output, int row, int col)
{
     int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
     int yIndex = blockDim.y * blockIdx.y + threadIdx.y;
     int index = (4*xIndex + yIndex * row*4);

     if(index < row*col*4){
          int lum = 0.11*input[index] + 0.59*input[index+1] + 0.3*input[index+2];

          output[index]=lum;
          output[index+1]=lum;
          output[index+2]=lum;
          output[index+3]=0;
     }
}
```

Brightness

```
// Scale each RGB value by the brightening factor


for(int y = 0; y < img->height(); y++) {
     for(int x = 0; x < img->width(); x++) {
          QRgb p = img->pixel(x, y);
          int r = qRed(p) * factor;
          int g = qGreen(p) * factor;
          int b = qBlue(p) * factor;
          // Clamp color values
          img->setPixel(x, y, qRgb(r, g, b));
     }
}
return img;
```

Contrast

```
// Scale each RGB value away from the average luminosity


for(int y = 0; y < img->height(); y++) {
     for(int x = 0; x < img->width(); x++) {
          QRgb p = img->pixel(x, y);
          int r = (1-factor)*average_lum + factor*(qRed(p));
          int g = (1-factor)*average_lum + factor*(qGreen(p));
          int b = (1-factor)*average_lum + factor*(qBlue(p));
          // Clamp color values
          img->setPixel(x, y, qRgb(r, g, b));
     }
}
return img;
```

Saturation

```
// Scale each RGB value away from its luminosity


for(int y = 0; y < img->height(); y++) {
     for(int x = 0; x < img->width(); x++) {
          QRgb p = img->pixel(x, y);
          float lum = (qRed(p) * 0.3) + (qGreen(p) * 0.59) + (qBlue(p) * 0.11);
          int r = (1-factor)*lum + factor*(qRed(p));
          int g = (1-factor)*lum + factor*(qGreen(p));
          int b = (1-factor)*lum + factor*(qBlue(p));
          // Clamp color values
          img->setPixel(x, y, qRgb(r, g, b));
     }
}
return img;
```

Crop

```
// Validate input dimensions and coordinates


// Copy pixels within the coordinates
for(int x = leftX; x < rightX; x++) {
     for(int y = leftY; y < rightY; y++)
          img->setPixel(x-leftX, y-leftY, img->pixel(x,y));
}
return img;
```

Rotate

```
// Determine new dimensions


// Rotate the image
for(int x = 0; x < newWidth; x++) {
     for(int y = 0; y < newHeight; y++) {
          float u = cos(-1*a)*(x-newxcenter) - sin(-1*a)*(y-newycenter)+width/2;
          float v = sin(-1*a)*(x-newxcenter) + cos(-1*a)*(y-newycenter)+height/2;
          // Perform Gaussian sampling to determine pixel color
          newImg->setPixel(x, y, Utility::GaussianSample(img, u, v, 0.6, 4));
     }
}
return newImg;
```

Scale

```
// Determine new dimensions


// Scale the image
for(int x = 0; x < newWidth; x++) {
     for(int y = 0; y < newHeight; y++) {
          float u = 1/factor*x;
          float v = 1/factor*y;
          // Perform sampling to determine pixel color
          newImg->setPixel(x, y, Utility::GaussianSample(img, u, v, 0.6,
2*factor));
     }
}
return newImg;
```

Blur

```
// Convolution mask
float mask[3][3] = { {1/16.0, 2/16.0, 1/16.0}, {2/16.0, 4/16.0,
2/16.0}, {1/16.0, 2/16.0, 1/16.0} };

for(int y = 0; y < height; y++) {
     for(int x = 0; x < width; x++) {
          int r = 0, g = 0, b = 0;
          for(int i = 0; i <= 2; i++) {
             for(int j = 0; j <= 2; j++) {
                    QRgb pixel = img->pixel(x+(j-1), y+(i-1));
                    r += mask[i][j]*qRed(pixel);
                    g += mask[i][j]*qGreen(pixel);
                    b += mask[i][j]*qBlue(pixel);
               }
          }
          // Clamp color values
     }
}
newImg->setPixel(x, y, qRgb(r, g, b));
return newImg;
```

Edge Detect

```
// Convolution mask
float mask[3][3] = { {-1.0, -1.0, -1.0}, {-1.0, 8.0, -1.0}, {-1.0,
-1.0, -1.0} };

for(int y = 0; y < height; y++) {
     for(int x = 0; x < width; x++) {
          int r = 0, g = 0, b = 0;
          for(int i = 0; i <= 2; i++) {
             for(int j = 0; j <= 2; j++) {
                    QRgb pixel = img->pixel(x+(j-1), y+(i-1));
                    r += mask[i][j]*qRed(pixel);
                    g += mask[i][j]*qGreen(pixel);
                    b += mask[i][j]*qBlue(pixel);
               }
          }
          // Clamp color values
     }
}
newImg->setPixel(x, y, qRgb(r, g, b));
return newImg;
```

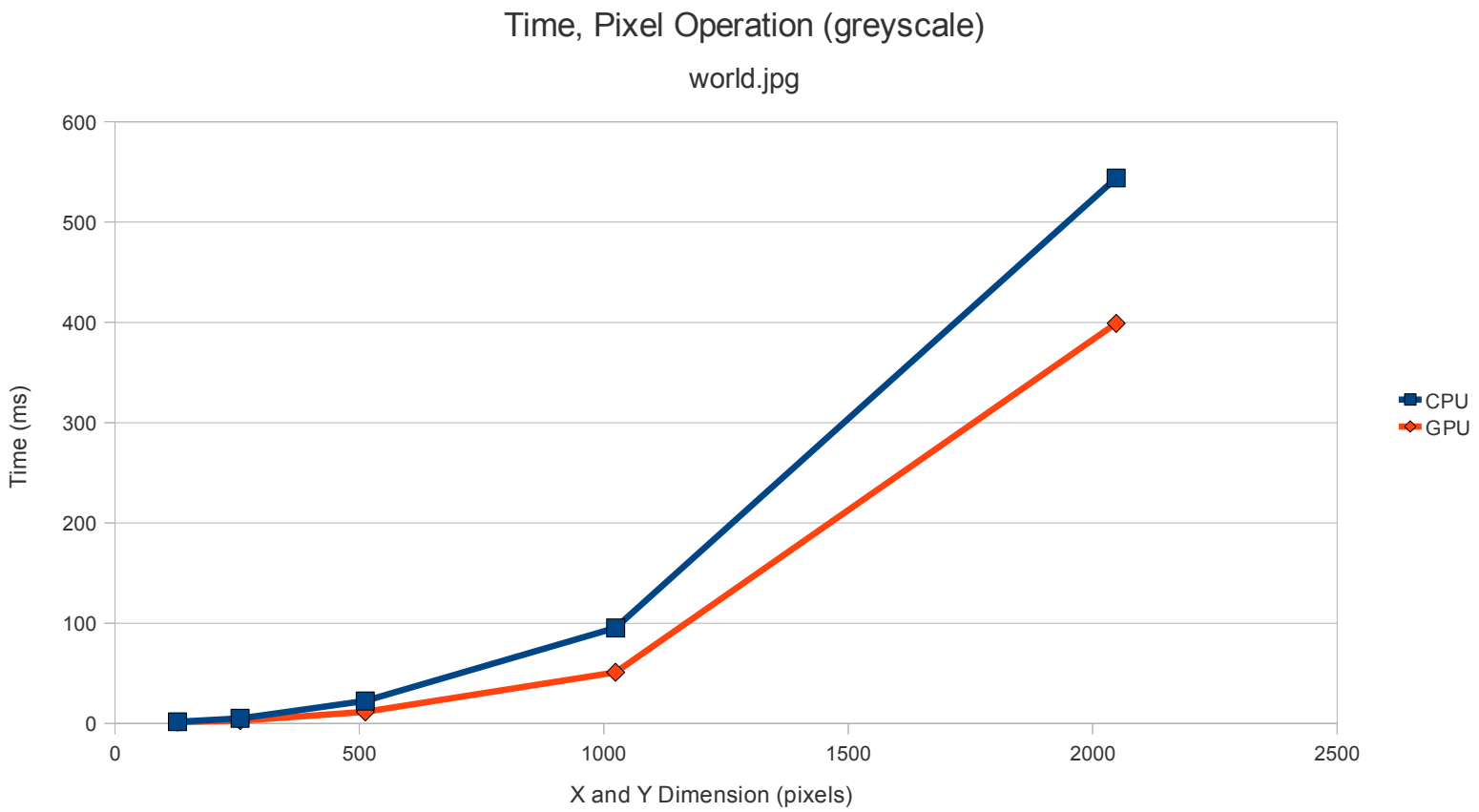
GPU: Edge Detect

```
extern "C" void CUedgeDetect(unsigned char* input, unsigned char* output, int row, int
col);

void CUedgeDetect(unsigned char* output, unsigned char* input, int row, int col)
{
    dim3 dimGrid(row/4+1, col/4+1);
    dim3 dimThreadBlock(16,16);

    float coeff[9]= {-1, -1, -1, \
                     -1,  8, -1, \
                     -1, -1, -1};
    float* CUcoeff;
    cutilSafeCall(cudaMalloc((void**)&CUcoeff, sizeof(float)*9));
    cutilSafeCall(cudaMemcpy(CUcoeff, coeff, sizeof(float)*9, cudaMemcpyHostToDevice));

    conv3x3<<<dimGrid, dimThreadBlock>>>(input, output, row, col, CUcoeff);
    cutilSafeCall(cudaFree(CUcoeff));
}
```

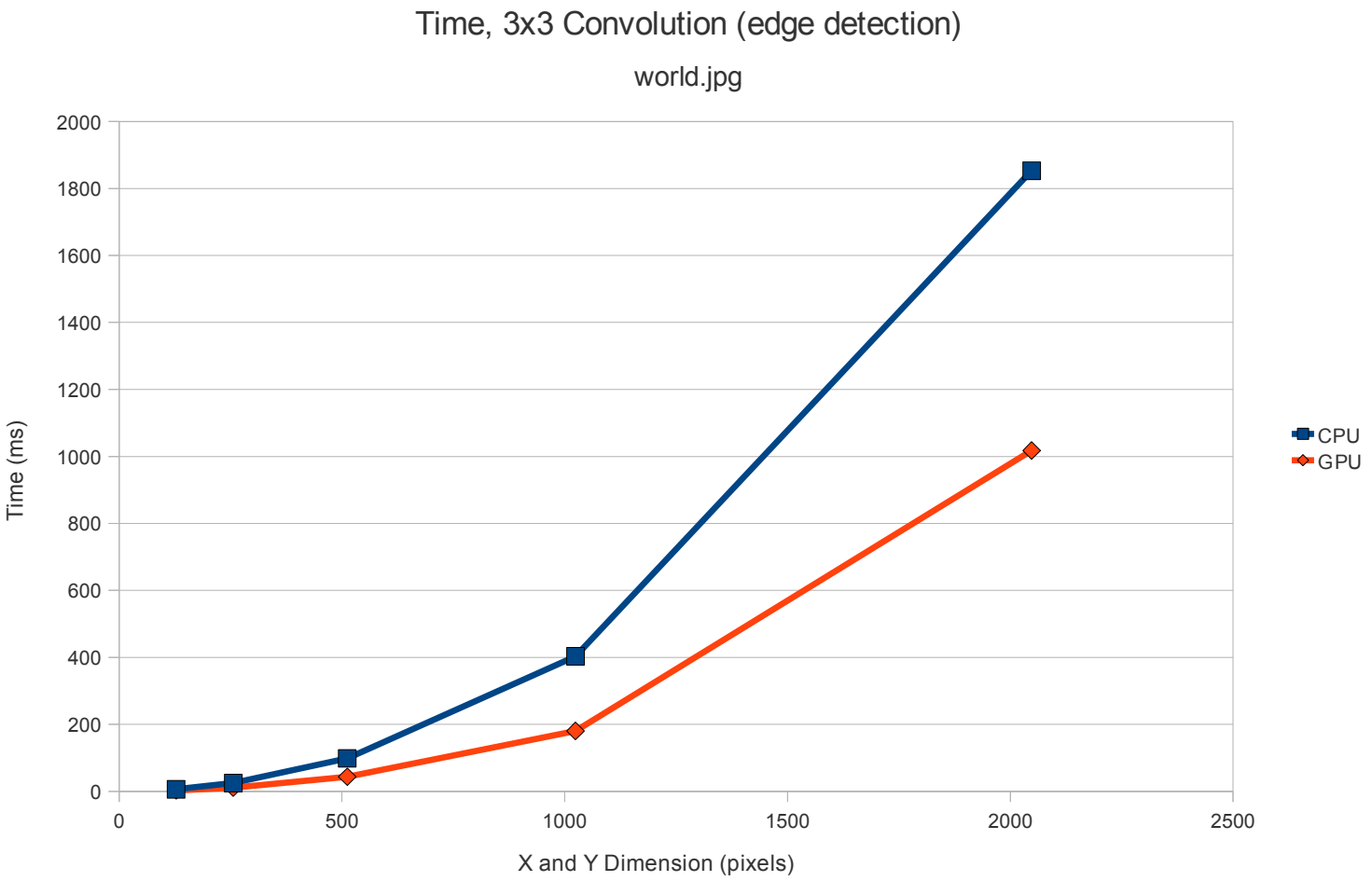
GPU: Edge Detect

```
__global__ void conv3x3(unsigned char* input, unsigned char* output, int row, int col,
float* kernel)
{
     int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
     int yIndex = blockDim.y * blockIdx.y + threadIdx.y;
     int index = (xIndex + yIndex * row*4);

     if(index < row*col*4){
          int i, j;
          float convSum=0;
          for(i=-1; i < 2; i++){
               for(j=-1; j < 2; j++){
                    if(-1 < (index+4*j)+(4*col*i) && (index+4*j)+(4*col*i) < row*col*4){
                    convSum += kernel[3*(i+1) + (j+1)]*input[(index+4*j)+(4*col*i)];
                    }
               }
          }
          output[index] = CLAMP(convSum);
     }
}
```

CPU vs GPU Speed Comparison



CPU vs GPU Speed Comparison



Decoding

QImage-provided support for .bmp, .gif, .jpg, .png, .pgm, and .tif decoding

Decoding implemented for custom .ppc codec

Image stored in 32-bit RGBA format internally

Encoding

QImage-provided support for .bmp, .jpg, .png, .tif encoding with quality setting option

Encoding implemented for custom .ppc codec

- Huffman Compression
- Arithmetic Compression
- Run Length Compression

Encoding

Sully.bmp: 329 Kb

None: 436 Kb

Run Length: 318 Kb

Huffman: 277 Kb

Arithmetic: 874 Kb

Huffman over Run Length: 260 Kb

Arithmetic over Run Length: 636 Kb

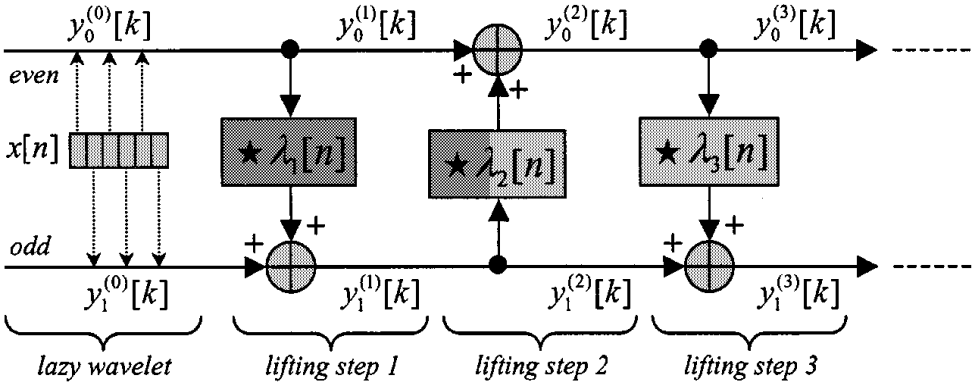
Arithmetic over Huffman: 554 Kb

Arithmetic over Huffman over Run Length: 528 Kb

.ppc Codec

```
void Encoder::write_ppc(QImage* img, QString filename, bool huffman, bool
arithmetic, bool runlength) {
     int mode = 4*runlength + 2*huffman + arithmetic;
     unsigned long numBytes = img->byteCount();
     unsigned char* byte_stream = img->bits();
     double* arithmetic_stream = NULL;

     if(runlength)  byte_stream = runlength_encode(byte_stream, &numBytes);
     if(huffman)    byte_stream = huffman_encode(byte_stream, &numBytes);
     if(arithmetic) arithmetic_stream = arithmetic_encode(byte_stream, &numBytes);

     FILE* output;
     if(!(output = fopen(filename.toStdString().c_str(), "w"))) { return; }
     fprintf(output, "%d %d %d %lu", mode, width, height, numBytes);
     if(!arithmetic)  fwrite(byte_stream, sizeof(unsigned char), numBytes, output);
     else             fwrite(arithmetic_stream, sizeof(double), numBytes, output);
     fclose(output);
}
```

Compression



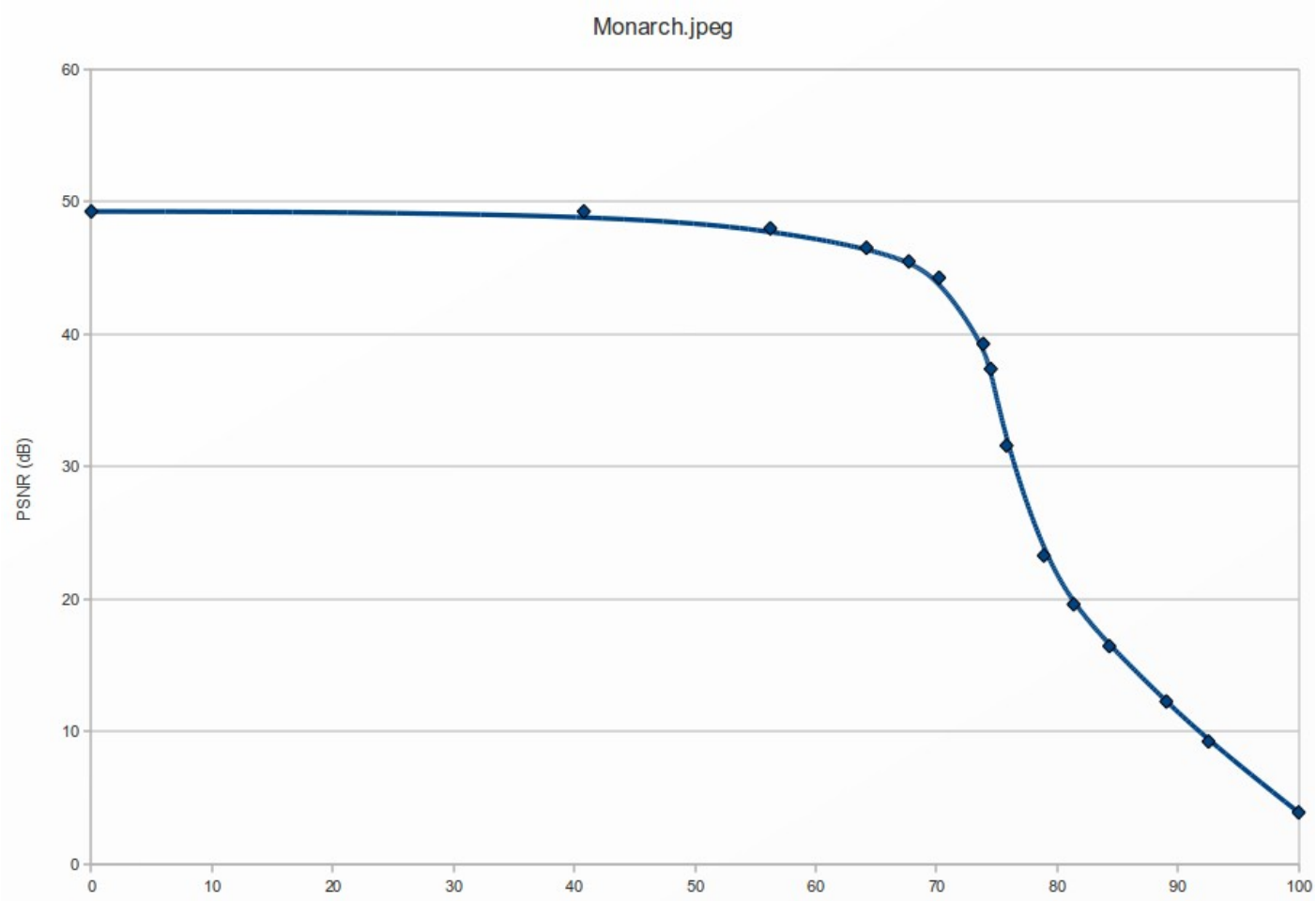
Discrete 9/7 Wavelet Transform, lifting implementation

- Used Grigory Pau's implementation, uses exact coefficients given by Taubman and Marcellin in JPEG2000 paper

  (http://www.embl.de/~gpau/misc/dwt97.c)

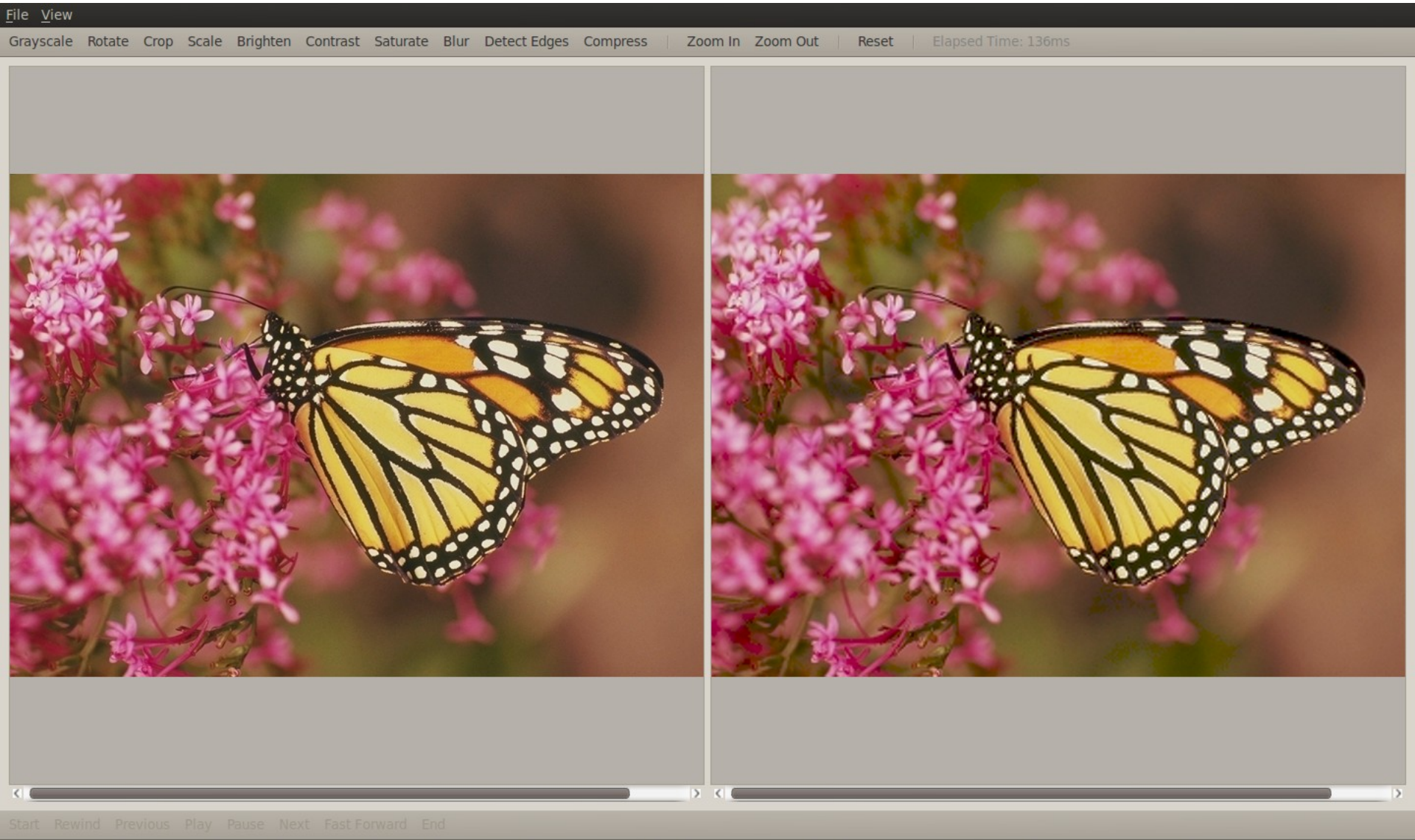
- CPU: lightly modified, GPU: heavily modified
- No color transform, still have to sort out colors
- One level transform
- Use piecewise continuous function to approximate threshold for zeroing out coefficients give a desired percent of coefficients zeroed out

Compression

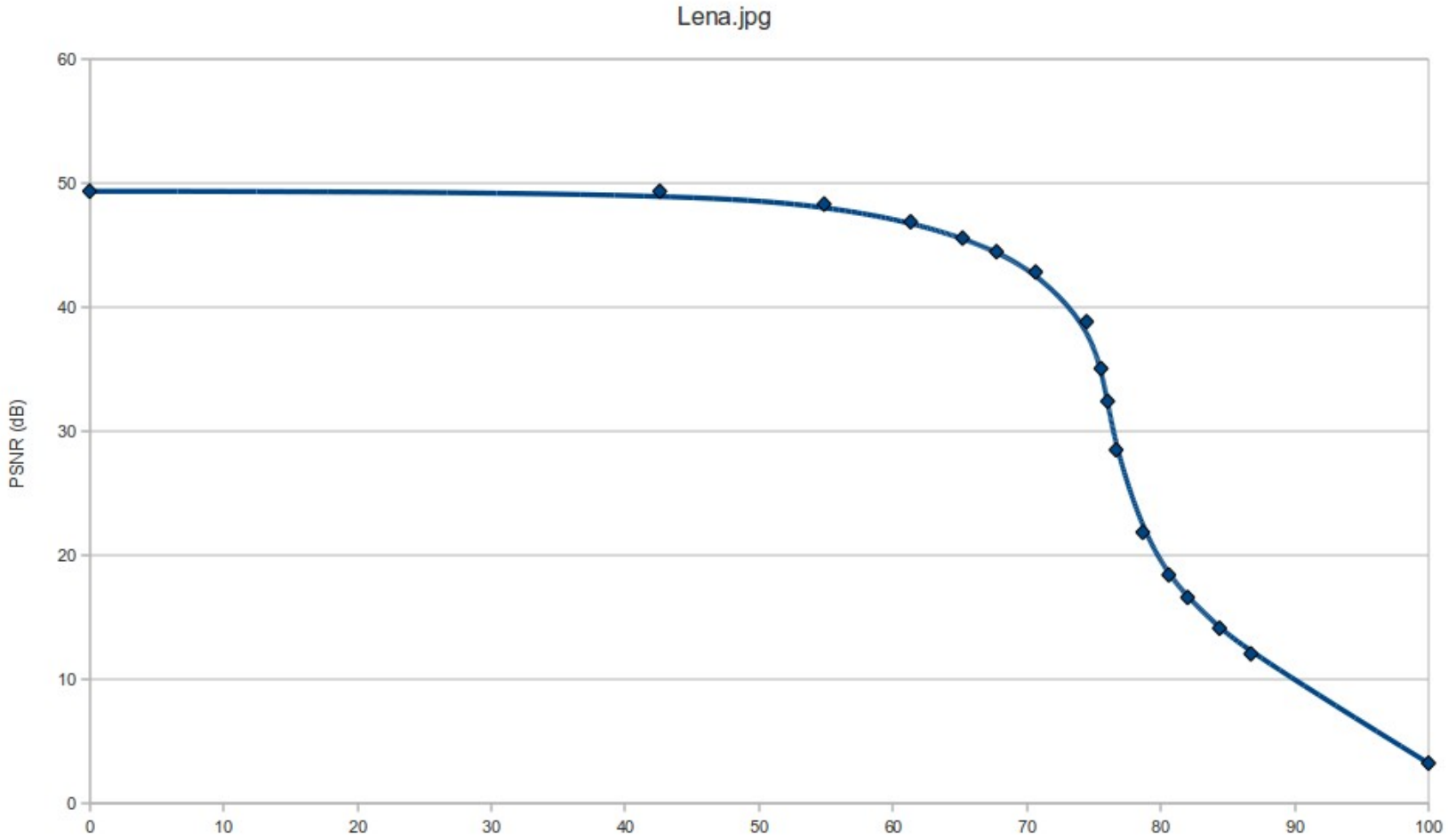


Adam Feinstein and H. Parker Shelton, 520.443, Johns Hopkins University

Compression

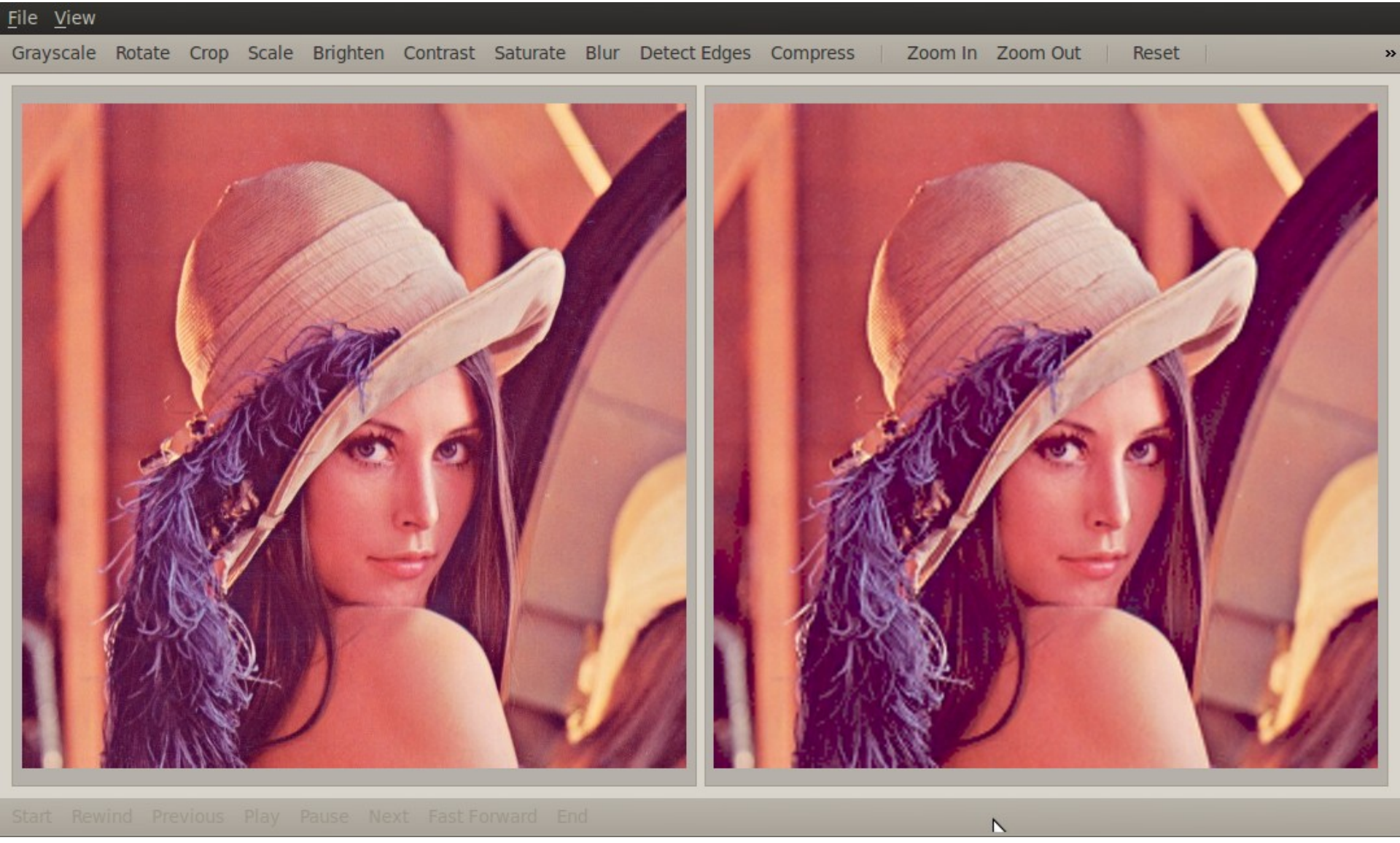


Compression

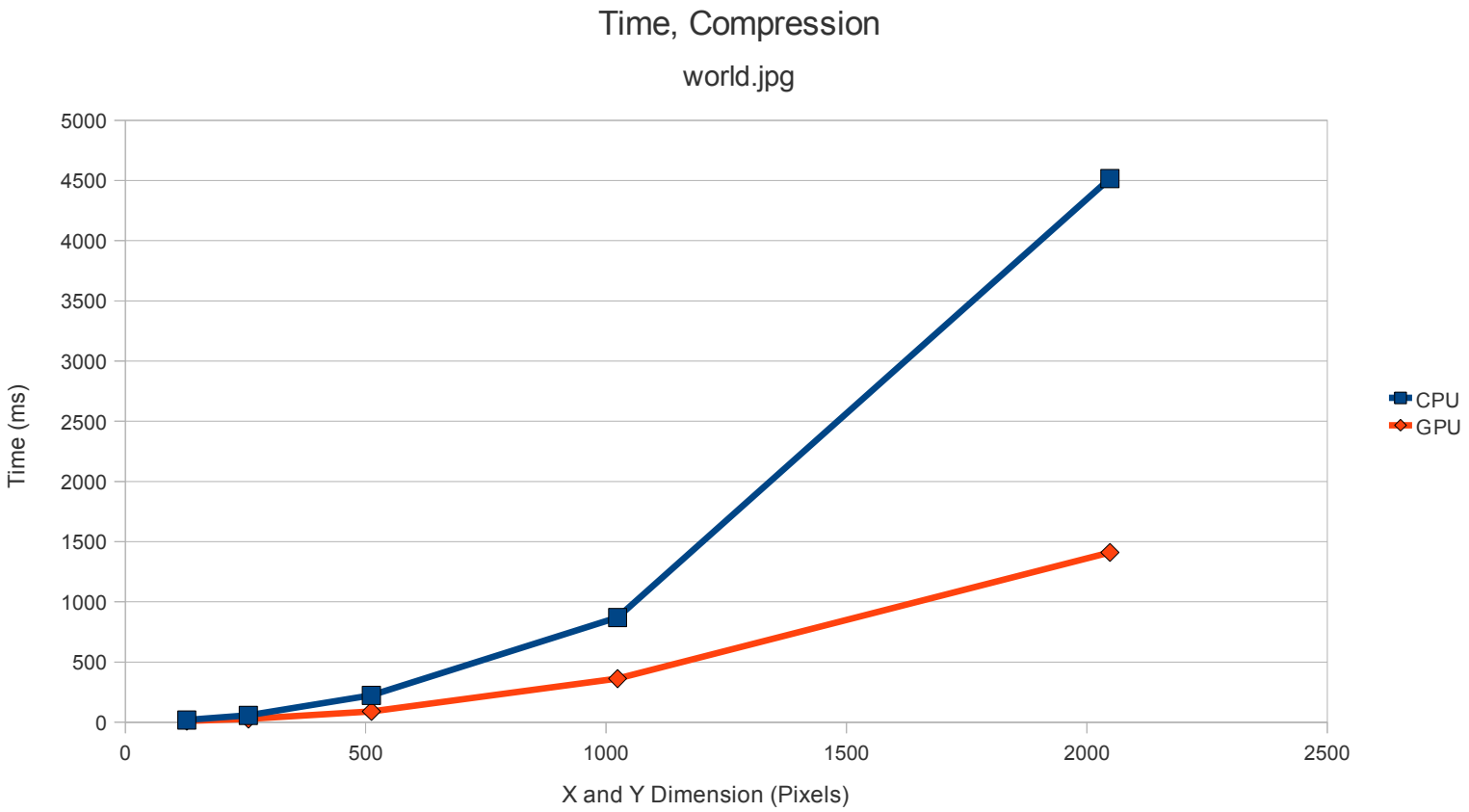


Adam Feinstein and H. Parker Shelton, 520.443, Johns Hopkins University

Compression



CPU vs GPU Speed Comparison



CPU vs GPU Speed Comparison

GPU Limitations:

- cudaMemcpy() still runs in linear time, necessary to move data onto and off of a GPU
- Resources used by operating system, failed to allocate space for a 4096x4096 image (64MB * 2)
- Wavelet transform doesn't use __shared__ memory

Questions?