MIC-GPU: High-Performance Computing for Medical Imaging on Programmable Graphics Hardware (GPUs)

Overview

Again, we will discuss two different programming models:
- graphics-style (first)
- GPGPU-style (second)

Both use the same underlying architecture (for example GeForce 8800 series)

Part 1

Graphics Style GPU Programming
- using fragment programs

GPU Image Processing Examples

Smoothing / low-pass filter:
- Gaussian blur
- 2-Pass using separable X, Y

Edge detection / high pass filter:
- using Sobel filter
- 3-Pass approach, Sobel_X, Sobel_Y, combine

Unsharp masking: combination of the above
- combination of results of both above filters

Recall: all operations are done for each pixel in parallel
- this is unlike traditional CPU programming, where pixels are operated on sequentially
**Smoothing / Low-Pass Filter**

Convolution with Gaussian Filter
- 2D filter → not practical
- 1D separable filter, 2 passes, about 10x-20x faster
  - 10x10=100 tex Lookups / pixel vs. 10/pixel for 2 passes=total 20/pixel

Practical GPU Implementation

1. Create a texture to store the convolution kernel.
   - Set size to 2X radius of the filter
   - Evaluate Gaussian kernel at each pixel
2. Create temp RenderBuffer to store intermediate result for 1st pass.
3. Run program GaussianBlur_separable along X-direction
   - Set glDrawBuffer(temp)
   - Loop to sample all points along X direction and use tex for weighting
4. Run program GaussianBlur_separable along Y-direction
   - Set glDrawBuffer(result)
   - Loop to sample all points along Y direction and use tex for weighting

**Main fragment program listing**

```c
float4 convolve_1D(
    uniform samplerRECT image : TEXUNIT0,   // the input image
    uniform samplerRECT kernel  : TEXUNIT1,   // the kernel texture
    uniform int kernel_width,             // kernel width
    uniform float2 texel_size,float2 pos : TEXCOORD0             // position in image
) : COLOR
{
    float4 c = 0;
    for(int x=0; x<kernel_width; x++) {
        float weight = texRECT(kernel, float2(x, 0)).r;
        c += texRECT(image, pos + (float2(x, x) ) * texel_size) * weight;
    }
    return c;
}
```

**Practical GPU Implementation**

1. Create a texture to store the convolution kernel.
   - Set size to 2x radius of the filter
   - Evaluate Gaussian kernel at each pixel
2. Create temp RenderBuffer to store intermediate result for 1st pass.
3. Run program GaussianBlur_separable along X-direction
   - Set glDrawBuffer(temp)
   - Loop to sample all points along X direction and use tex for weighting
4. Run program GaussianBlur_separable along Y-direction
   - Set glDrawBuffer(result)
   - Loop to sample all points along Y direction and use tex for weighting

**Main fragment program listing**

```c
float4 convolve_1D(   
    uniform samplerRECT image : TEXUNIT0,   /* the input image */
    uniform samplerRECT kernel  : TEXUNIT1,   /* the kernel texture */
    uniform int kernel_width,             /* kernel width */
    uniform float2 texel_size,             /* texel size */
    float2 pos : TEXCOORD0             /* position in image */
) : COLOR
{
    float4 c = 0;
    for(int x=0; x<kernel_width; x++) {
        float weight = texRECT(kernel, float2(x, 0)).r;
        c += texRECT(image, pos + (float2(x, x) ) * texel_size) * weight;
    }
    return c;
}
```

**Edge Detection / High Pass Filter**

Use simple 2D Sobel masks

\[
G_x = \begin{bmatrix}
    -1 & 0 & +1 \\
    -2 & 0 & +2 \\
    -1 & 0 & +1
\end{bmatrix} * A \quad \text{and} \quad 
G_y = \begin{bmatrix}
    +1 & +2 & +1 \\
    0 & 0 & 0 \\
    -1 & -2 & -1
\end{bmatrix} * A
\]

Three pass algorithm:

1. Create tempX, tempY, and result RenderBuffers for intermediate results.
2. Run program sobel along X-direction using GX mask
   - Set glDrawBuffer(tempX)
   - Weigh all 3x3 neighboring points according to mask
3. Run program sobel along Y-direction using GY mask
   - Set glDrawBuffer(tempY)
   - Weigh all 3x3 neighboring points according to mask
4. Run program sobel_combine
   - Set glDrawBuffer(result)
   - For every pixel, result = sqrt(pixelFromX^2 + pixelFromY^2)

**Main fragment program listing**

```c
fragout applyMask( float4 TexCoords : TEXCOORD0,
    float4 WinPos : WPOS,
    uniform samplerRECT inputTexture : TEXUNIT0,
    uniform float3x3 mask)
{
    fragout OUT;
    float sum=0;
    float2 bottomLeft=WinPos.xy - float2(-1,-1);
    int x,y;
    for(x=0; x<3; x++){
        for(y=0; y<3; y++){
            int x2 = x+bottomLeft.x;
            int y2 = y+bottomLeft.y;
            sum+=mask[x][y] * texRECT(inputTexture, float2(x2, y2));
        }
    }
    OUT.col.rgb = sum;
    return OUT;
}
```

**Edge Detection / High Pass Filter**

Use simple 2D Sobel masks

\[
G_x = \begin{bmatrix}
    -1 & 0 & +1 \\
    -2 & 0 & +2 \\
    -1 & 0 & +1
\end{bmatrix} * A \quad \text{and} \quad 
G_y = \begin{bmatrix}
    +1 & +2 & +1 \\
    0 & 0 & 0 \\
    -1 & -2 & -1
\end{bmatrix} * A
\]

Three pass algorithm:

1. Create tempX, tempY, and result RenderBuffers for intermediate results.
2. Run program sobel along X-direction using GX mask
   - Set glDrawBuffer(tempX)
   - Weigh all 3x3 neighboring points according to mask
3. Run program sobel along Y-direction using GY mask
   - Set glDrawBuffer(tempY)
   - Weigh all 3x3 neighboring points according to mask
4. Run program sobel_combine
   - Set glDrawBuffer(result)
   - For every pixel, result = sqrt(pixelFromX^2 + pixelFromY^2)
Unsharp Masking

Unsharp masking is a combination of filters.

Here, \[ U_{\text{mask}} = g*I + (1+\alpha)(I - g*I) \]

To implement, we can use the results of the smoothing operation and use a second pass to apply the combinations.

GPGPU Style GPU Programming
- using CUDA

Convolution (2D)

Each thread will compute one output pixel.
**Implementation**

- Load a block of the image into shared memory → load-threads
  - block size is determined by available shared memory
- Compute convolution sum for all block pixels (exclusive apron pixels) → compute-threads
  - then write result to shared memory

**Thread Management**

For large masks, will get large aprons
- will lead to many idle (load)-threads once computing begins
- leads to under utilization of the processors → poor performance

Reduce this problem by loading more than one pixel per thread
- for example, a column of pixels, which is easy to compute an index for
- ideally have the same number of load threads than compute threads

**Separable Filters**

Nevertheless, a pixel may be loaded 9 times in total due to overlapping block-apron tiles

Again, replacing a 2D convolution into two orthogonal 1D convolutions can help
- will shrink the number of reloads to 6 → but effect is not large due to the relatively small tile sizes
- however, convolution along x eliminates the need for aprons in the y-direction and convolution along y eliminates the need for aprons in the x-direction
  - percentage of apron data much smaller
  - more pixels can be loaded for processing
- limitations are in thread block size, not in shared memory
  - need to perform more computation within a thread
  - process more pixels within a thread (arithmetic intensity ↑)

**Memory Access Conflicts**

- Constant Memory (contains filter mask): no conflict → all threads will access the same location (storing the same filter coefficient)

- Shared Memory (contains data):
  - consecutive threads access consecutive memory locations → since there are 16 banks there is no shared memory bank conflict
Unrolling Loops

Original loop:
```c
for(int k = -KERNEL_RADIUS; k <= KERNEL_RADIUS; k++)
    sum += data[sharMemPos + k] * d_Kernel[KERNEL_RADIUS - k];
```

Loop body has very few operations
- overhead by loop/branching is relatively high

Solution: loop unrolling
```c
```

Results in 2-fold performance increase

Memory Coalescence

Bandwidth to off-chip ("device") DRAM is much higher than on a host CPU memory.

However, in order to achieve high memory throughput, the GPU seeks to **coalesce** accesses from multiple threads into a single memory transaction:
- if all threads within a warp (32 threads) simultaneously read consecutive words then single large read of the 32 values can be performed at optimum speed
- however, if 32 random addresses are read, then only a fraction of the total DRAM bandwidth can be achieved, and performance will be much lower.

Summary: CUDA Optimizations

Coalesce memory operations
- coordinate reads (by a half-warp)
- greatly improves throughput (can yield speedups of >10)

Hide latency
- more threads/block → better memory latency hiding
- however, more threads/block → fewer registers/thread
- choose threads/block as multiple of warp size
- minimum: 64 threads/block, 128-256 better choice (experiment!)

Prevent shared memory bank conflicts
- conflict-free shared memory as fast as registers

Metrics
- Performance measured in GFlops,
To Probe Further

NVIDIA CUDA Zone:
- lots of information and code examples
- NVIDIA CUDA Programming Guide

GPGPU community:
- http://www.gpgpu.org
- user forums, tutorials, papers

CUDA occupancy calculator available at:

Course Schedule

1:30 – 2:00: Introduction
2:00 – 2:30: GPU architecture, programming model, and programming facilities
2:30 – 3:00: GPU programming examples (image processing)

Coffee Break

3:30 – 4:00: CT reconstruction pipeline components
4:00 – 4:30: GPU-acceleration of individual components
4:30 – 5:00: Various CT reconstruction pipelines, load balancing and load estimation
5:00 – 5:30: Reconstruction visualization and final remarks