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

SPIE Medical Imaging

GPU Programming Examples

Klaus Mueller

Computer Science
Center for Visual Computing
Stony Brook University

STONY BROOK Center for Visual Computing

Overview

SPIE Medical Imaging

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

SPIE Medical Imaging

Graphics Style GPU Programming

- using fragment programs

GPU Image Processing Examples

SPIE Medical Imaging

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

SPIE Medical Imaging

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

Smoothing / Low-Pass Filter

SPIE Medical Imaging

Main fragment program listing

```
float4 convolve_1D(
            uniform samplerRECT image  : TEXUNIT0,   // the inpu
            uniform samplerRECT kernel  : TEXUNIT1,   // the k
            uniform int kernel_width,           // kern
            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;
}
```

Input Texture0 for Image
Use Texture1 to store the pre-computed kernel

Texel Size is set to (1,0) for the first pass, along X and (0,1) for the second pass, along Y

Lookup the appropriate Kernel Value and use as weight

Lookup the appropriate Image Value and add to the weighted sum

Edge Detection / High Pass Filter

SPIE Medical Imaging

Use simple 2D Sobel masks

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

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)

Edge Detection / High Pass Filter

SPIE Medical Imaging

Main fragment program listing

```
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++){
                  sum+=mask[x][y] * texRECT(inputTexture, float2(x,y)+bottomLeft).x;
                  }
         }
         OUT.col.rgba=sum;
         return OUT;
}
```

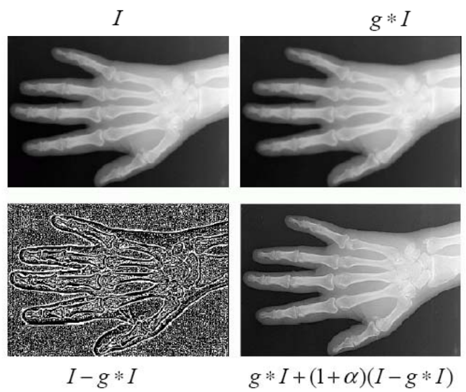
Mask is provided as 3x3 parameter from the calling program.
float Gx[]= {-1, 0, +1,
-2, 0, +2,
-1, 0, +1};

Unsharp Masking

Unsharp masking is a combination of filters

Here,
Umask= $g*I+(1+\alpha)(I-g*I)$

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



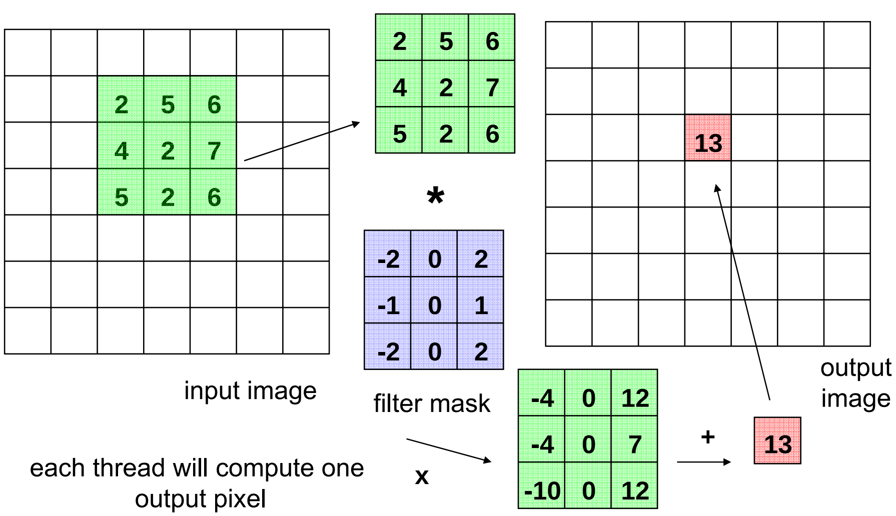
I → Original Image
$g*I$ → Smoothed with Gaussian
α → Edge Enhancement Parameter

Part 2

GPGPU Style GPU Programming

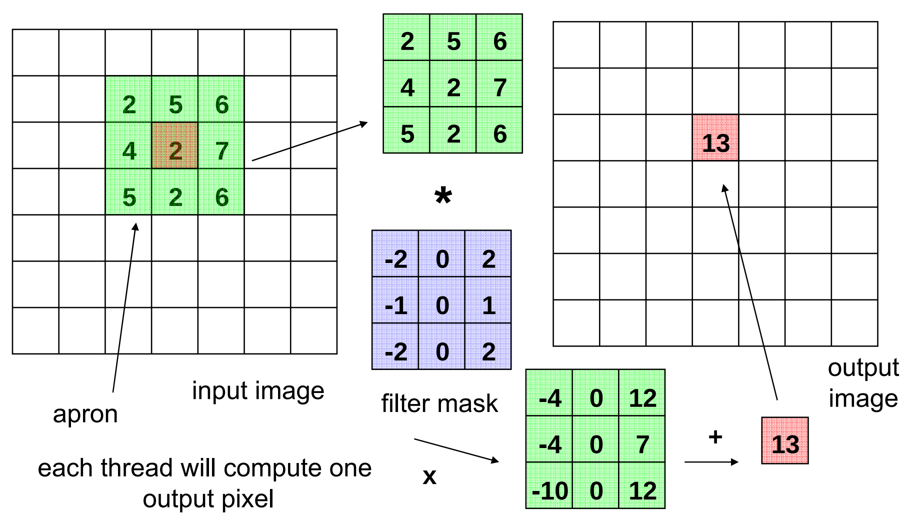
- using CUDA

Convolution (2D)



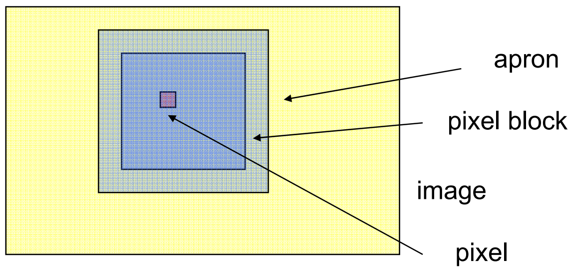
each thread will compute one output pixel

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

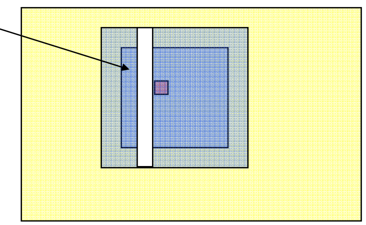
Thread Management

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:

```
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

```
sum = data[sharMemPos - 1] * d_Kernel[2]
    + data[sharMemPos + 0] * d_Kernel[1]
    + data[sharMemPos + 1] * d_Kernel[0];
```

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,

Summary: CUDA Optimizations

Maximize arithmetic intensity

- unroll loops
- often computing is better than caching (data transfer costly)

Optimize memory coherence

- exploit spatial locality
- coalesce

Distribute load well

- keep processors equally busy
- minimize idle threads

Maximize occupancy

- minimize latency
- optimize number of threads running on each multiprocessor

To Probe Further

NVIDIA CUDA Zone:

- http://www.nvidia.com/object/cuda_home.html
- lots of information and code examples
- NVIDIA CUDA Programming Guide

GPGPU community:

- http://www.gpgpu.org
- user forums, tutorials, papers
- good source: Supercomputing 2007 CUDA tutorial
  http://www.gpgpu.org/sc2007/

CUDA occupancy calculator available at:

- http://news.developer.nvidia.com/2007/03/cuda_occupancy_.html

Course Schedule

1:30 – 2:00:	Introduction
2:00 – 2:30:	GPU architecture, programming model, and programming facilties
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