
The GPGPU Programming Model



Aaron Lefohn

Institute for Data Analysis and Visualization
University of California, Davis

Overview

- Data-parallel programming basics
- The GPU as a data-parallel computer
- “Hello World” GPGPU Example
- Programming model detail
- Conclusions



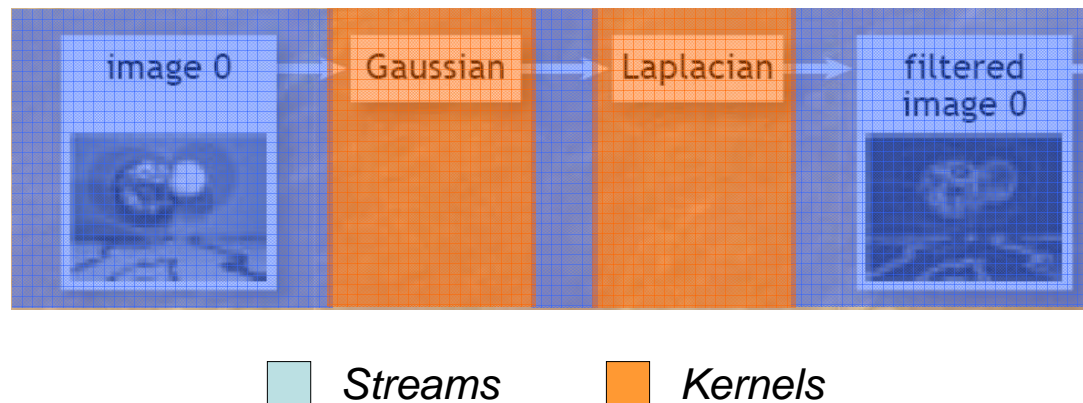
Data-Parallel Programming Basics

- What is a data-parallel program?
 - Explicitly expresses data dependencies
 - Exposes parallelism
- Stream programming is data-parallel model
 - Stream programs are dependency graphs
 - Kernels are graph nodes
 - Streams are edges flowing between kernels



Stream Program Basics

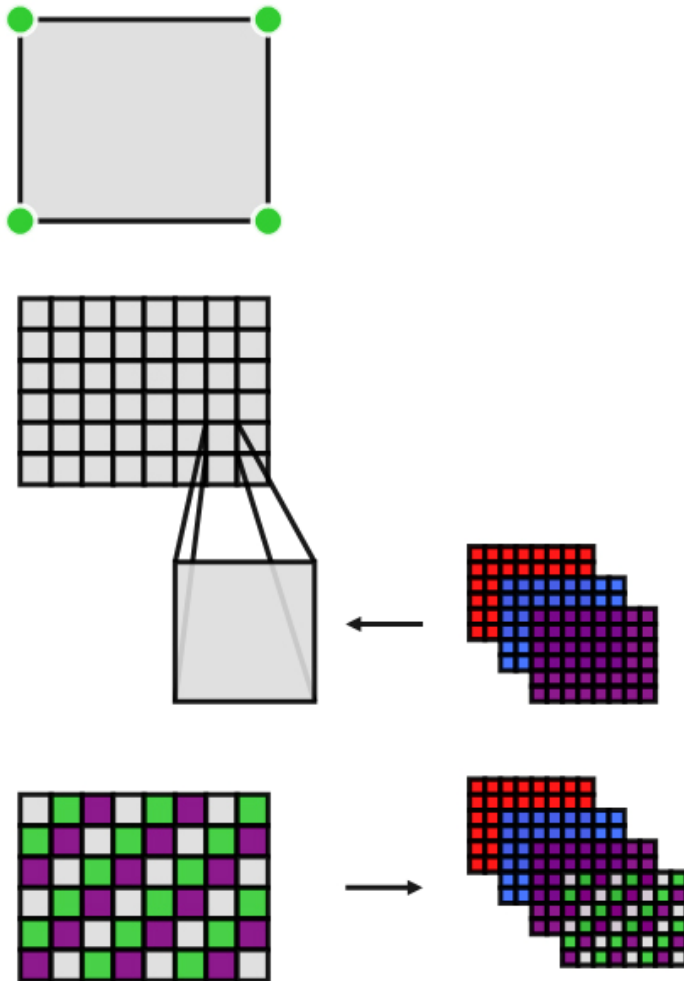
- The stream programming model
 - Split algorithm into kernels based on dependencies
 - Example: Image processing



Dally et al., “**Stream Processors: Programmability with Efficiency,**”
ACM Queue, March 2004, pp. 52-62
<ftp://cva.stanford.edu/pub/publications/spqueue.pdf>



GPU as a Stream Processor

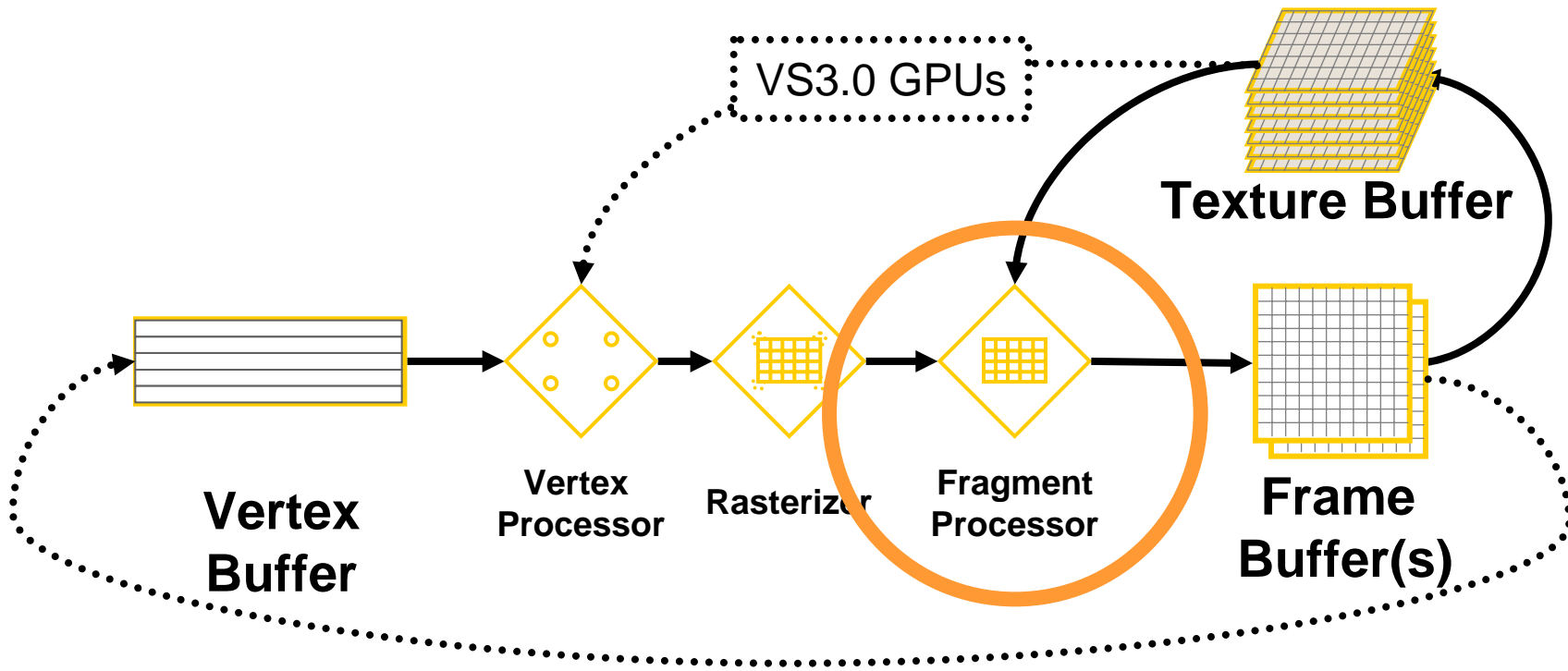


- Draw a screen-sized quad
- Run kernel over each fragment (fragment program)
- Read stream data from textures
- Write results to frame buffer

Figure courtesy of John Owens

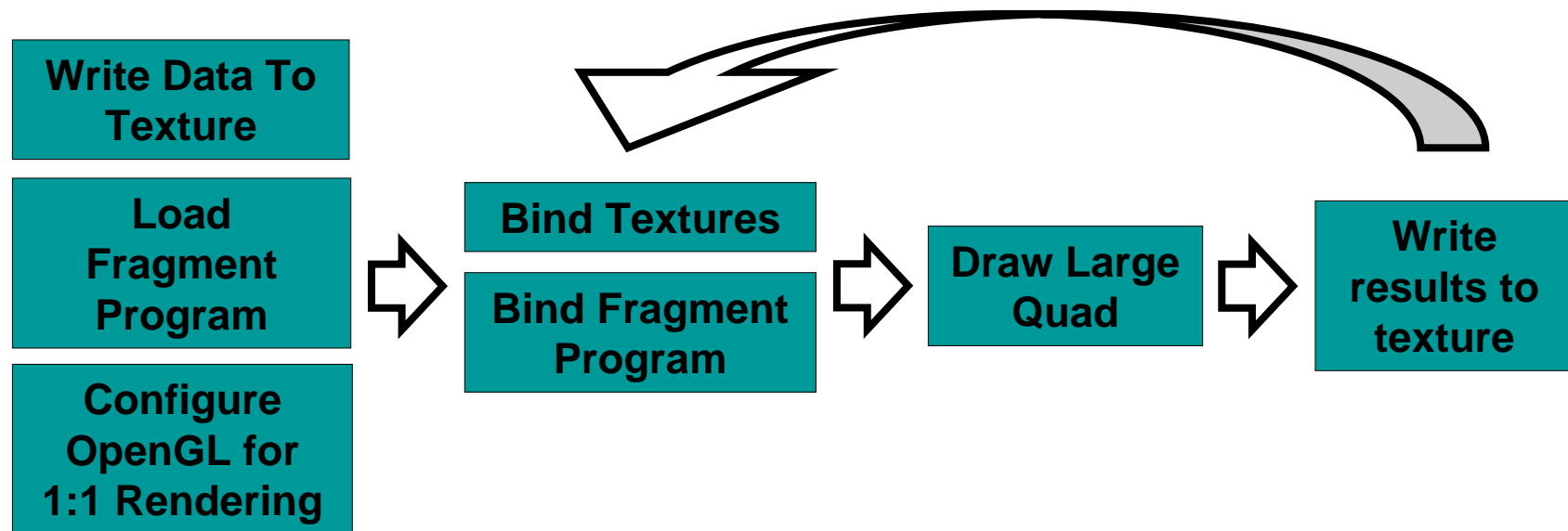


Modern Graphics Pipeline



GPU as a Stream Processor

- Streams → Textures
- Kernels → Fragment program
- `forEach` execution → Draw single large quad



“Hello World“ GPGPU Example

- 3 x 3 Image processing convolution
- CPU version

```
image = loadImage( WIDTH, HEIGHT );
blurImage = allocZeros( WIDTH, HEIGHT );

for (x=0; x < WIDTH; x++)
    for (y=0; y < HEIGHT; y++)
        for (i=-1; i <= 1; i++)
            for (j=-1; j <= 1; j++)
                float w = computeWeight(i,j);
                blurImage[x][y] += w * image[x+i, y+j];
```



“Hello World“ GPGPU Example

- GPU Version

- 1) Load `image` into texture

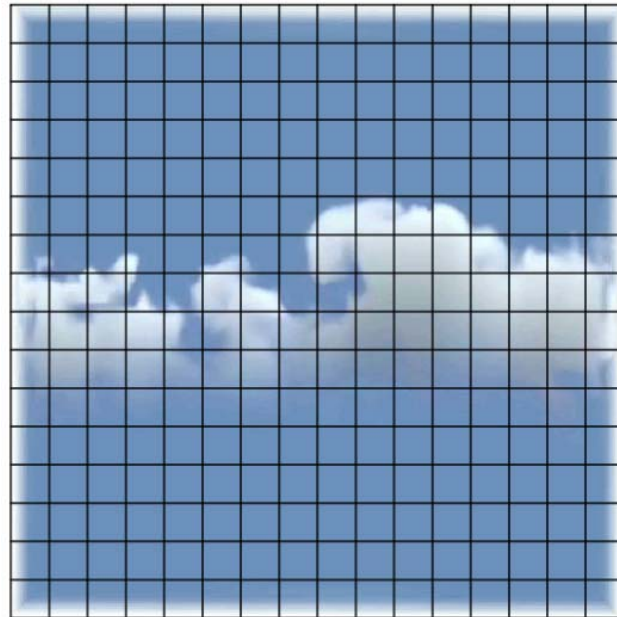


Figure courtesy of Mark Harris

- 2) Create `blurImage` texture to hold result



“Hello World“ GPGPU Example

- GPU Version

- 3) Load fragment program (kernel)

Example shown in Cg

```
float4 blurKernel( uniform samplerRECT image,
                  float2      winPos : WPOS,
                  out float4  blurImage )
{
    blurImage = float4(0,0,0,0);

    for (i=-1; i <= 1; i++) {
        for (j=-1; j <= 1; j++) {
            float2 texCoord = winPos + float2(i,j);
            float  w         = computeWeight(i,j);
            blurImage += w * texRECT( image, texCoord );
        }
    }
}
```



“Hello World“ GPGPU Example

- GPU Version

- 4) Configure OpenGL to draw 1:1
No projection or rescaling

```
glMatrixMode( GL_PROJECTION );  
glLoadIdentity();  
gluOrtho2D(0, 1, 0, 1);  
glViewport(0, 0, WIDTH, HEIGHT );  
glMatrixMode( GL_MODELVIEW );  
glLoadIdentity();
```

- 5) Bind `image` and `blurKernel` (texture and fragment program)
- 6) Bind `blurImage` as render target



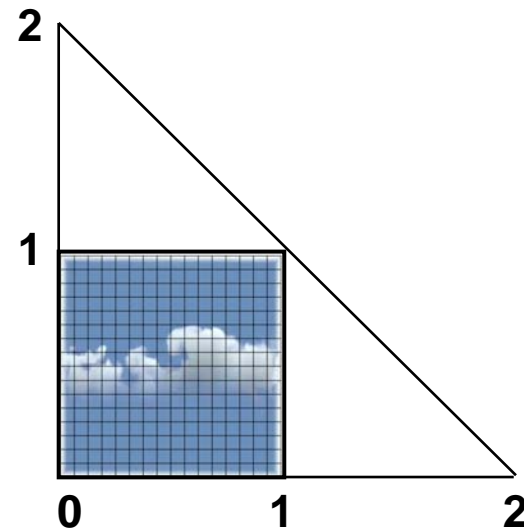
“Hello World“ GPGPU Example

- GPU Version

7) Execute kernel on each stream element

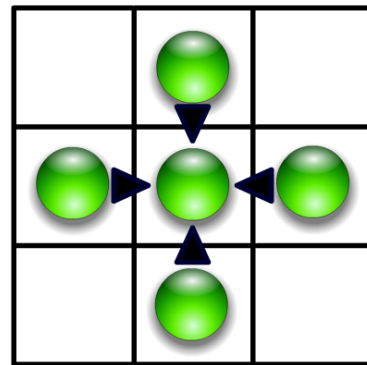
Draw quad of size [WIDTH x HEIGHT]

```
glBegin( GL_TRIANGLES );  
    glVertex2f(0, 0);  
    glVertex2f(2, 0);  
    glVertex2f(0, 2);  
glEnd();
```



“Hello World“ GPGPU Example

- What happened?
 - `blurKernel` executed on each element of `image`
 - Rendering replaced outer two loops of CPU version
 - `blurKernel` performed *gather* operation at each element



Gather

- Result (`blurImage`) was written to framebuffer / texture



“Hello World“ GPGPU Example

- Get the source code for GPGPU examples
 - <http://www.gpgpu.org/developer/>
 - http://download.nvidia.com/developer/SDK/Individual_Samples/samples.html
 - gpgpu_fluid
 - gpgpu_disease
 - gpu_particles
 - http://www.ati.com/developer/sdk/RadeonSDK/Html/Samples/OpenGL/HW_Image_Processing.html



Questions?

- GPU as stream processor?
- Stream programming model?
- “Hello World” example?



Moving On...

- Programming model detail
- Intro to handling missing features



Why Are Stream Programs Fast?

- Explicit parallelism
 - No communication between stream elements
 - Kernels cannot write to input streams
 - Kernels cannot have writable static/global variables
 - Stream elements are independent
- Explicit memory locality
 - Temporary values in kernel are local
 - Stream program expresses producers-consumers
 - Hide cost of memory access with parallelism



GPU Computational Primitives

- Operations available in kernel

- Read-only memory (input streams) Texture sampler
- Random access read-only (gather) Texture sampler
- Per-data-element interpolants Varying registers
- Temporary storage (no saved state) Local registers
- Read-only constants Constant registers
- Write-only memory (result streams) Render-to-texture
- Floating-point ALUops



GPU Computational Primitives

- What's missing?
 - No stack
 - No heap
 - No integer or bitwise operations
 - No scatter ($a[i] = b$)
 - No reduction operations (max, min, sum)
 - Data-dependent conditionals
 - Limited number of outputs

- Why missing?
 - Lack of demand from games
 - Early in GPU evolution as general data-parallel processor
 - Stream programming model constraints



GPU Computational Primitives

- Handling missing features
 - We'll explain how to emulate
 - Scatter
 - Global (reduction) operations
 - Conditionals



Emulating Scatter

- Scatter

```
i = foo();  
a[i] = bar();
```

- Solution 1

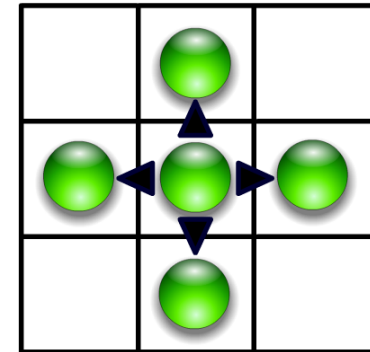
- Transform scatter algorithm into gather algorithm
- See Ian's "Tips and Tricks" for more details

- Solution 2

- Do actual scatter
- Vertex processor can scatter points
 - Render points instead of large triangle
 - Render-to-texture with vertex-texture-reads (PS 3.0)
 - Or render-to-vertex-array

- Problem

- Drawing a point for each data element is slow



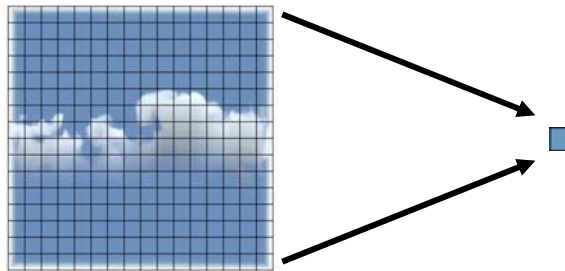
Scatter



Emulating Reduction Operations

- Reductions

- Operations that require all data elements
- max, min, sum, norm, etc.



Cloud figure courtesy of Mark Harris

- Solution

- Perform repeated gathers until only single data value left
- $\log(\text{WIDTH})$ gather operations (assuming $\text{WIDTH} == \text{HEIGHT}$)

- Problem

- Extra passes can be costly



Conditionals In a SIMD World

- Current fragment processors are SIMD
 - Vertex processors are MIMD (PS 3.0)
- SIMD execution and conditionals are at odds
 - SIMD model assumes all data elements processed identically
 - Conditional execution breaks this assumption
- Solutions
 - 1) Statically resolve conditional outside of kernel
 - 2) Disable “false“ stream elements
 - 3) Improved fragment hardware?

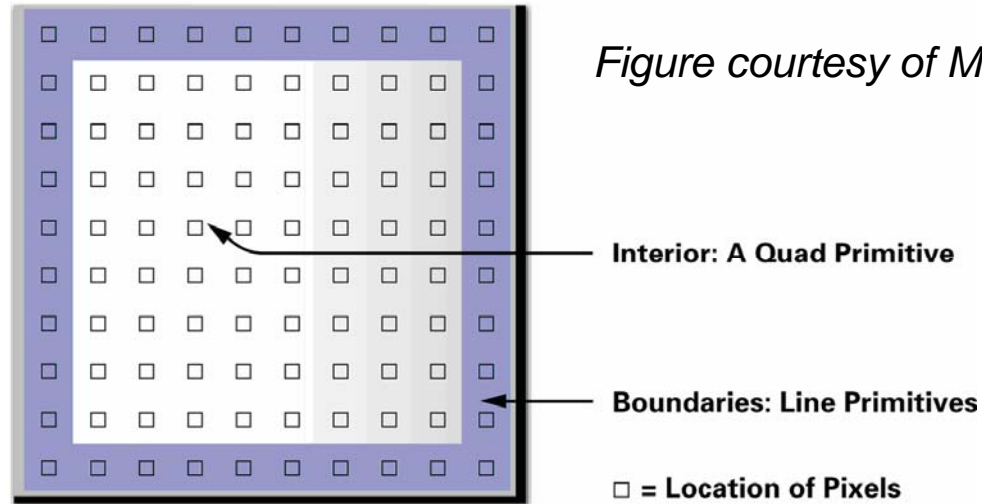


Conditionals in a SIMD World

- Solutions

1) Can decision be made before fragment processor?

Static branch resolution with substreams



Trick “discovered” by Lefohn, Harris, and Goodnight Simultaneously in 2003



Conditionals in a SIMD World

- Solution
 - 2) Disable “false” stream elements: Occlusion query
- Idea
 - Occlusion query reports the number of fragments that passed the depth test
 - Useful when number of loop iterations is data-dependent
 - Kernel kills fragments that do not need further processing
 - CPU continues to issue render until all fragments are killed



Conditionals in a SIMD World

- Solution
 - 3) Disable “false” stream elements: Early depth cull
- Idea
 - Modern GPUs can kill fragments before kernel execution
 - Kernel sets z-value to control whether or not execution occurs
- Problem
 - Conditionals must be block-coherent
 - More about this in Ian’s “Tips and Tricks” talk



Conditionals in a SIMD World

- Solution
 - 4) Improved fragment hardware?
 - MIMD hardware
 - NVIDIA GeForce 6-series (NV4x) vertex processors
 - SIMD-with-conditional-support
 - NVIDIA GeForce 6-series (NV4x) fragment processor
 - Uniform branches are a win if more than ~5 instructions
 - Varying branches must be coherent across hundreds of pixels
 - Only 10s x 10s pixels—Very useful in many cases
- Problem
 - Do we want more SIMD processors or fewer MIMD processors?



Conditionals in a SIMD World

- Conclusions
 - Conditionals are tough with today's GPUs
 - Best case is when conditional can be statically resolved and removed from computational kernel
 - Future GPUs will most likely fully support conditionals
 - Solution must not interfere with parallelism
 - MIMD?
 - SIMD with "Conditional Streams?"
 - Kapasi et al., Micro 33, 2000



Conclusions

- GPGPU computational basics

- Textures → Storage for data streams
- Fragment program → Computational kernel
- Render pass → `forEach` loop over data stream

- Coming next...

- Ian Buck
- Data-parallel languages for GPGPU programming
 - Express data-parallel programs more elegantly than `glBegin()...glEnd()`



References

- ATI Developer web site, <http://www.ati.com/developer/>
- GPGPU Developer web site, <http://www.gpgpu.org/developer>
- N. Goodnight, C. Woolley, G. Lewin, D. Luebke, G. Humphreys, "A Multigrid Solver for Boundary Value Problems Using Programmable Graphics Hardware," Graphics Hardware 2003
- M. Harris, W. Baxter, T. Scheuermann, A. Lastra, "Simulation of Cloud Dynamics on Graphics Hardware," Graphics Hardware 2003
- Hillis et al., "Data Parallel Algorithms," Comm. ACM, 29(12), December 1986
- Kapasi et al., "Efficient Conditional Operations for Data-parallel Architectures," In Proc. of the 33rd Ann. Int'l Symp. on Microarchitecture, pages 159--170, 2000
- A. Lefohn, J. Kniss, C. Hansen, R. Whitaker, "A Streaming Narrow-Band Algorithm: Interactive Deformation and Visualization of Level Sets," IEEE Transactions on Visualization and Computer Graphics 2004
- NVIDIA Developer web site, <http://developer.nvidia.com/page/home>

