King Abdullah University of Science and Technology

KAUST

CS 380 - GPU and GPGPU Programming
Lecture 19: GPU Texturing 6; Stream Computing and GPGPU

Markus Hadwiger, KAUST

Reading Assignment #11 (until Nov 16)

Read (required):

- Programming Massively Parallel Processors book, 3rd edition

  Chapter 5 (Performance Considerations) [was Chap. 6 in 2nd ed.]

Read (optional):

- Linear algebra operators for GPU implementation of numerical algorithms, Krueger and Westermann, SIGGRAPH 2003

  https://dl.acm.org/doi/10.1145/882262.882363

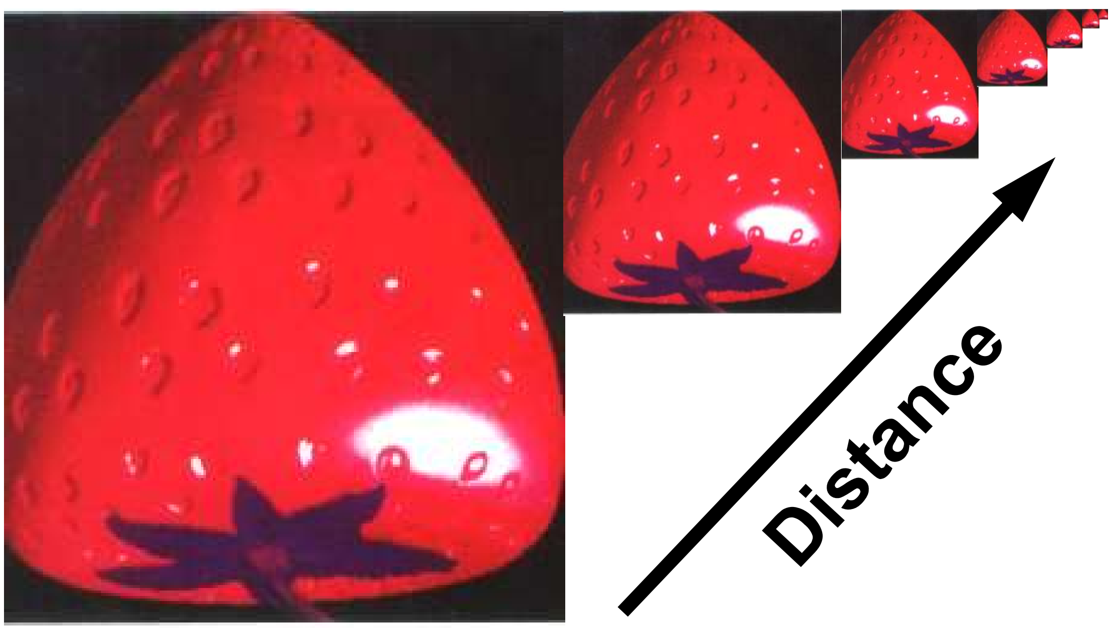
- A Survey of General-Purpose Computation on Graphics Hardware (2007)

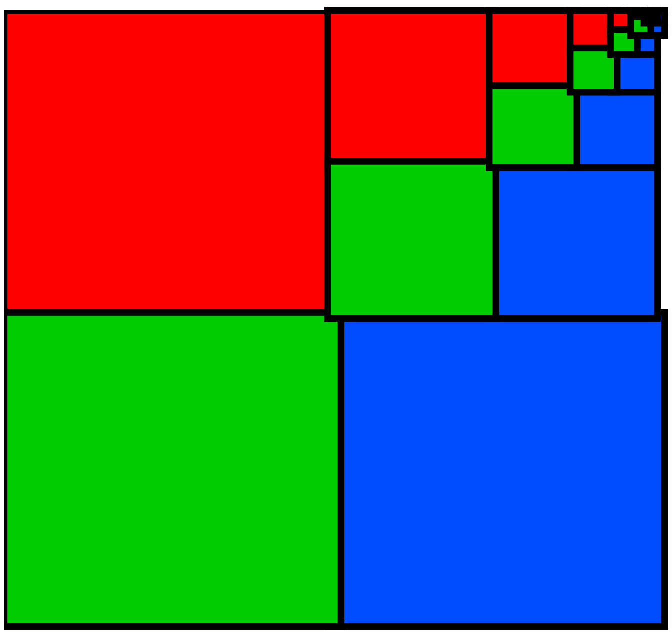
  https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2007.01012.x

Texture Minification

Texture Anti-Aliasing: MIP Mapping

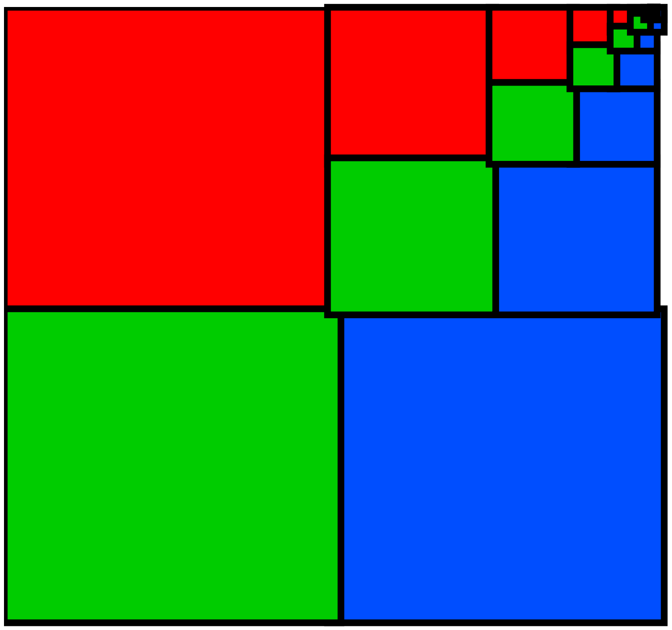
- MIP Mapping (“Multum In Parvo”)
  - Texture size is reduced by factors of 2 (*downsampling* = "many things in a small place")
  - Simple (4 pixel average) and memory efficient
  - Last image is only ONE texel





Texture Anti-Aliasing: MIP Mapping

- MIP Mapping ("Multum In Parvo")
  - Texture size is reduced by factors of 2 (*downsampling* = "many things in a small place")
  - Simple (4 pixel average) and memory efficient
  - Last image is only ONE texel



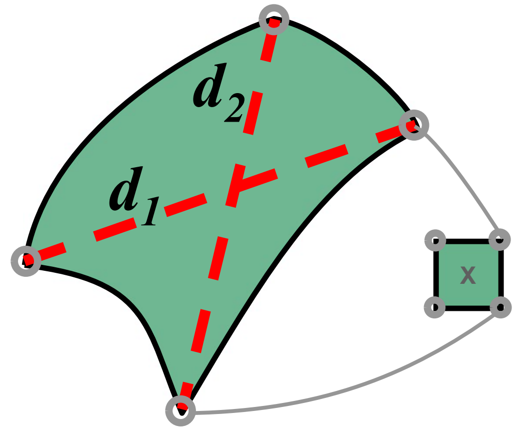
geometric series:

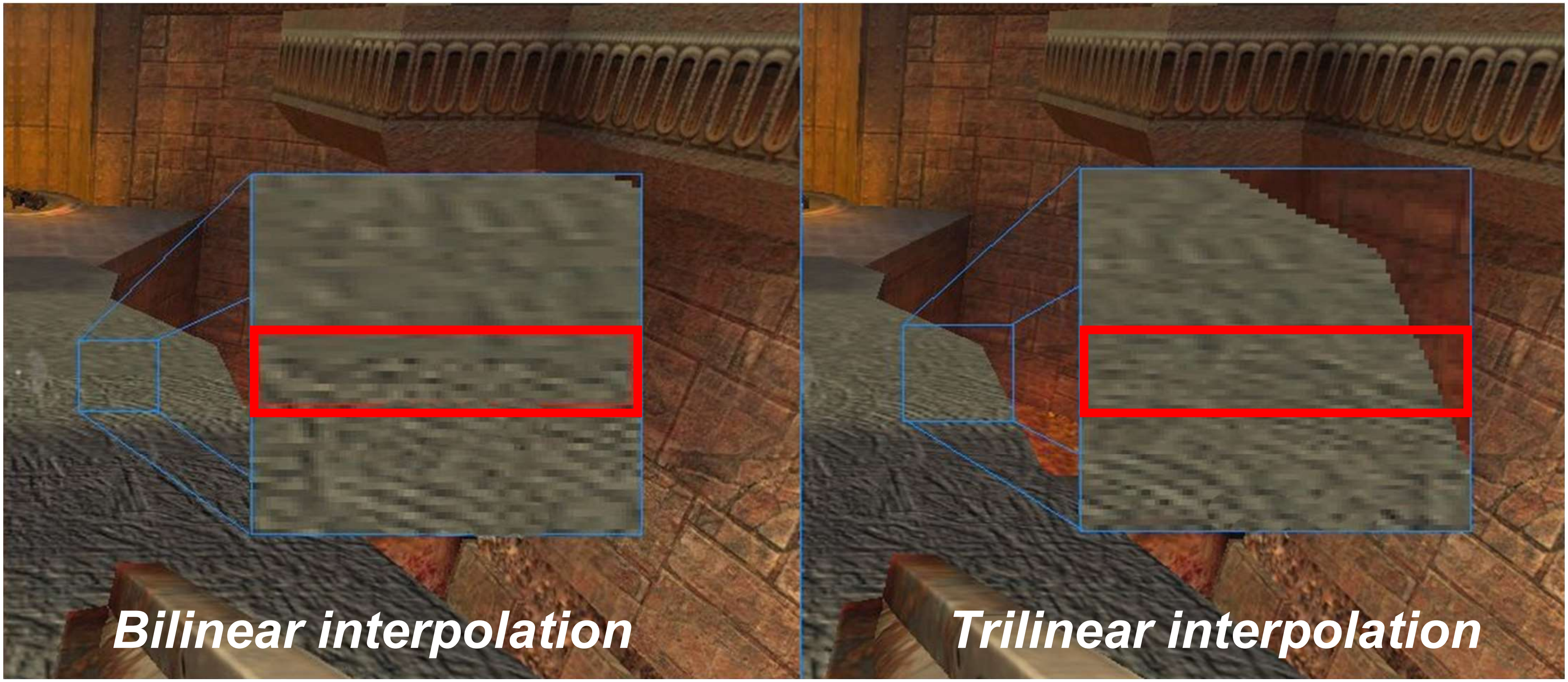
$$a + ar + ar^2 + ar^3 + \cdots + ar^{n-1} =$$

$$= \sum_{k=0}^{n-1} ar^k = a \left(\frac{1 - r^n}{1 - r} \right)$$

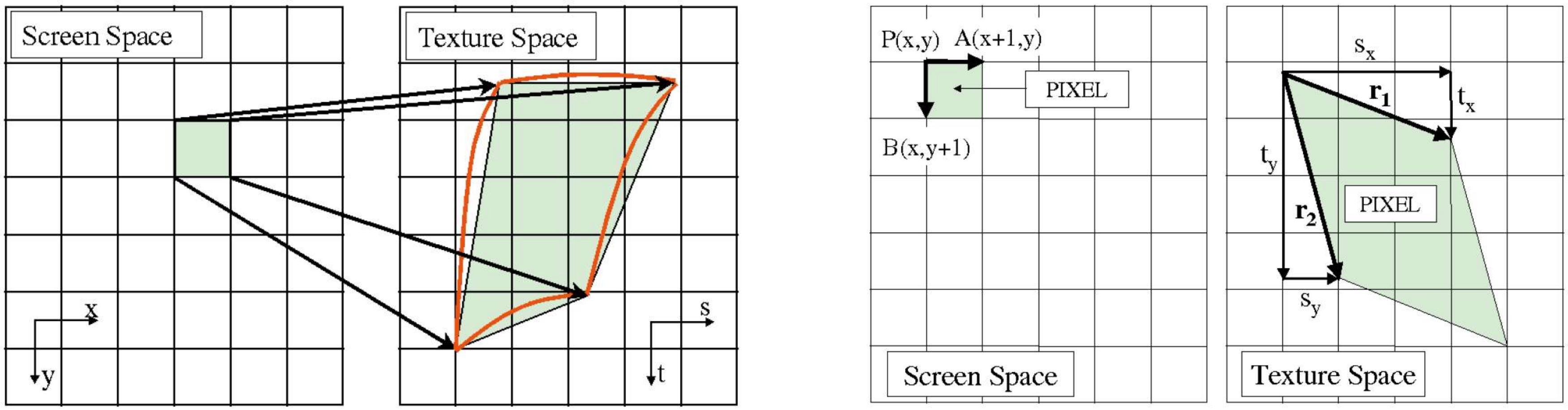
Texture Anti-Aliasing: MIP Mapping

- MIP Mapping Algorithm
- $D := ld(max(d_1,d_2))$
- T_0 := value from texture $D_0 = trunc\ (D)$ "Mip Map level"
  - Use *bilinear interpolation*





MIP-Map Level Computation



- Use the partial derivatives of texture coordinates with respect to screen space coordinates
- This is the Jacobian matrix

$$\begin{pmatrix} \partial u/\partial x & \partial u/\partial y \\ \partial v/\partial x & \partial v/\partial y \end{pmatrix} = \begin{pmatrix} s_x & s_y \\ t_x & t_y \end{pmatrix}$$

- Area of parallelogram is the absolute value of the Jacobian determinant (the *Jacobian*)

MIP-Map Level Computation (OpenGL)

- OpenGL 4.6 core specification, pp. 251-264

(3D tex coords!)

$$\lambda_{base}(x, y) = \log_2[\rho(x, y)]$$

$$\rho = \max\left\{\sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial w}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2 + \left(\frac{\partial w}{\partial y}\right)^2}\right\}$$

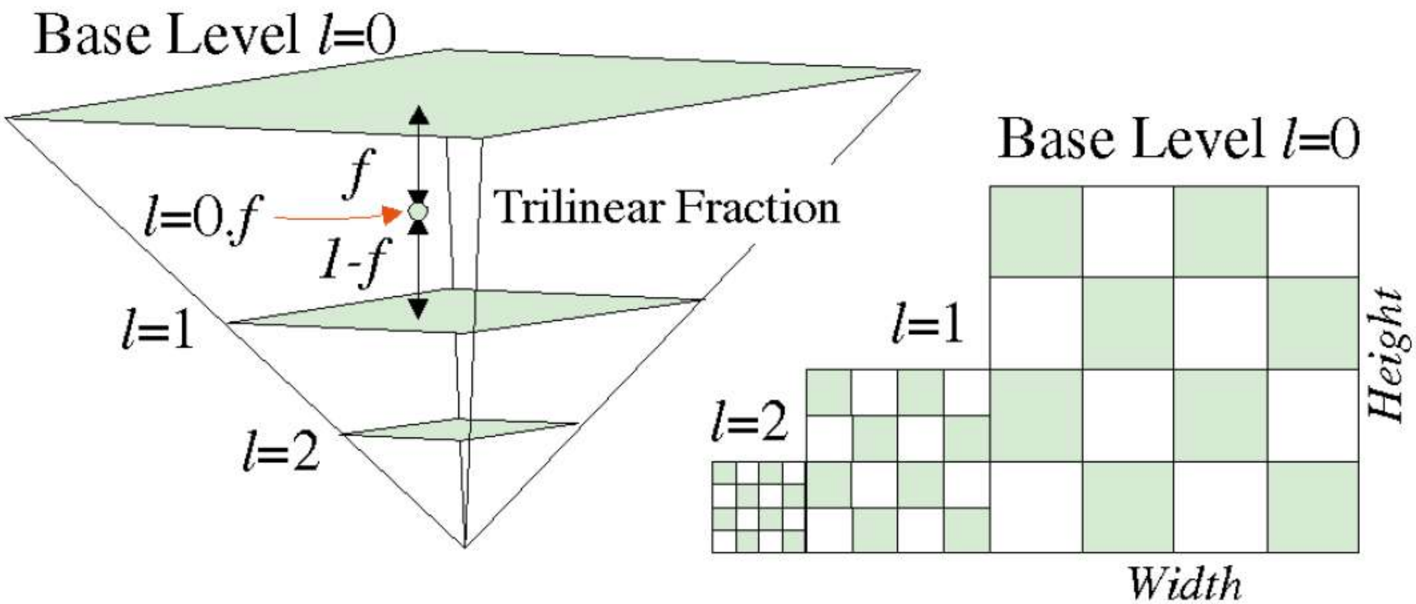
Does not use area of parallelogram but greater hypotenuse [Heckbert, 1983]

- Approximation without square-roots

$$m_u = \max\left\{\left|\frac{\partial u}{\partial x}\right|, \left|\frac{\partial u}{\partial y}\right|\right\} \quad m_v = \max\left\{\left|\frac{\partial v}{\partial x}\right|, \left|\frac{\partial v}{\partial y}\right|\right\} \quad m_w = \max\left\{\left|\frac{\partial w}{\partial x}\right|, \left|\frac{\partial w}{\partial y}\right|\right\}$$

$$\max\{m_u, m_v, m_w\} \leq f(x, y) \leq m_u + m_v + m_w$$

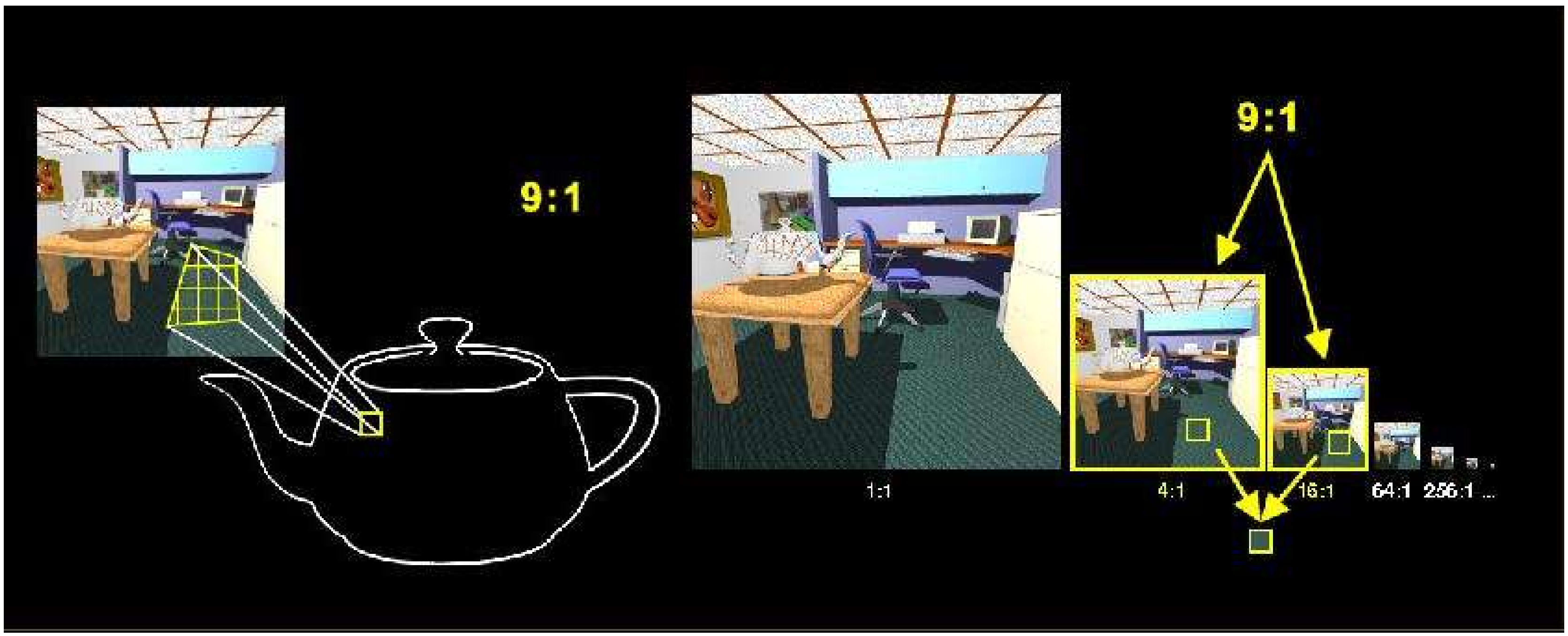
MIP-Map Level Interpolation



- Level of detail value is fractional!
- Use fractional part to blend (lin.) between two adjacent mipmap levels

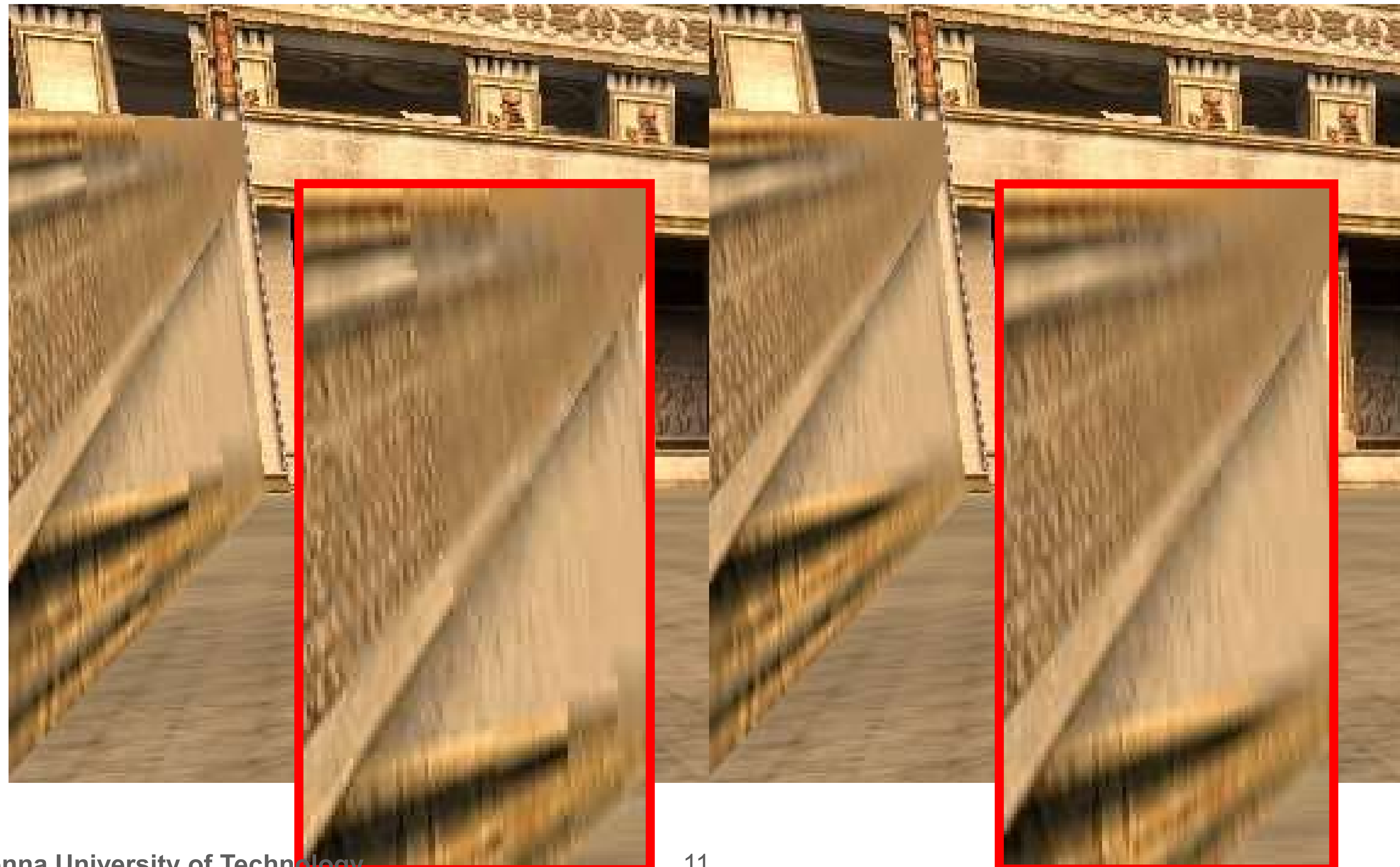
Texture Anti-Aliasing: MIP Mapping

- Trilinear interpolation:
  - T_1 := value from texture $D_1 = D_0+1$ (bilin.interpolation)
  - Pixel value := $(D_1–D)\cdot T_0 + (D–D_0)\cdot T_1$
    - Linear interpolation between successive MIP Maps
  - Avoids "Mip banding" (but doubles texture lookups)



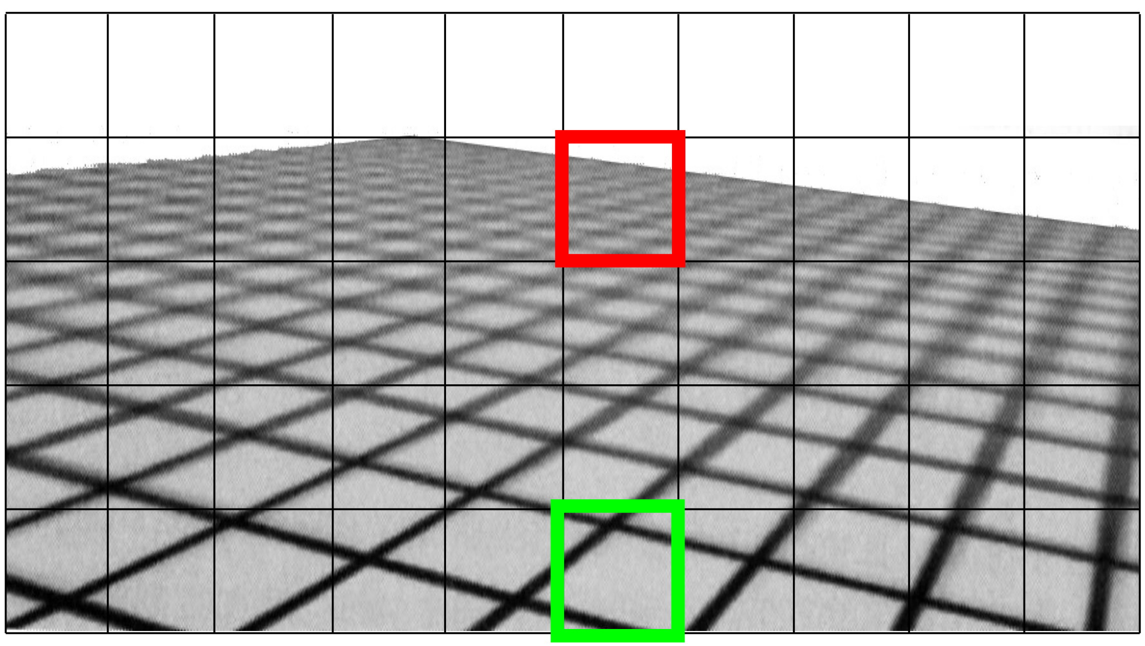
Texture Anti-Aliasing: MIP Mapping

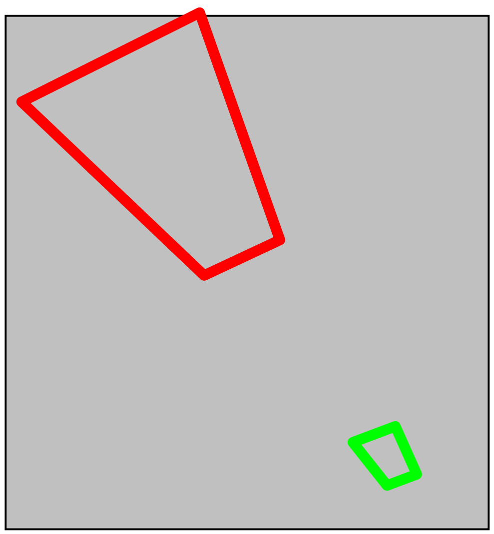
- Other example for bilinear vs. trilinear filtering



Anti-Aliasing: Anisotropic Filtering

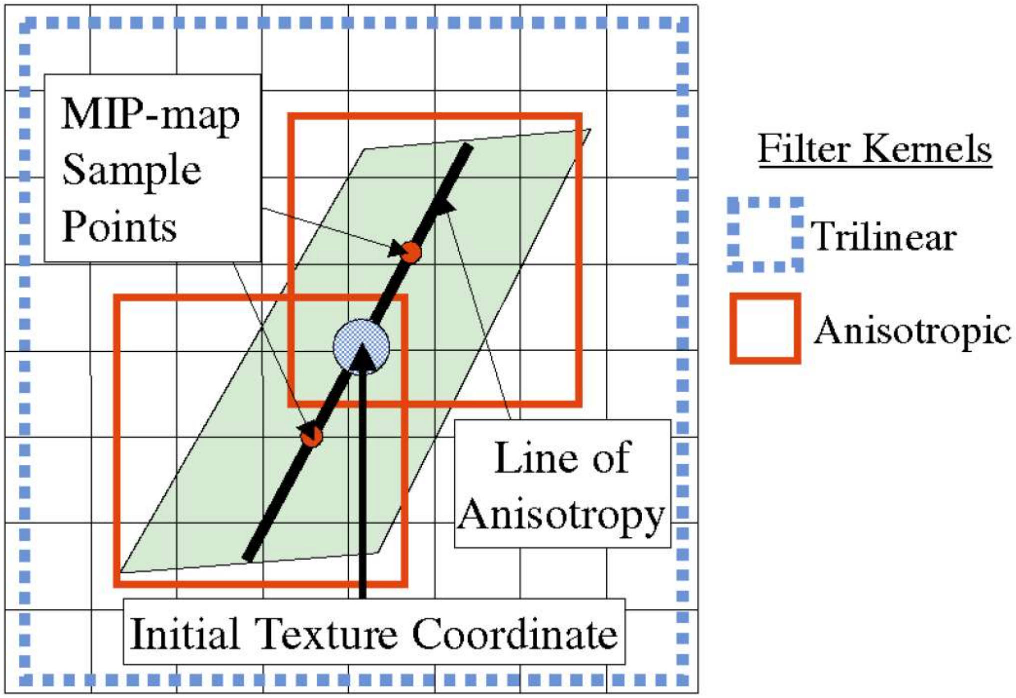
- Anisotropic filtering
  - View-dependent filter kernel
  - Implementation: *summed area table*, "*RIP Mapping*", *footprint assembly*, *elliptical weighted average* (EWA)





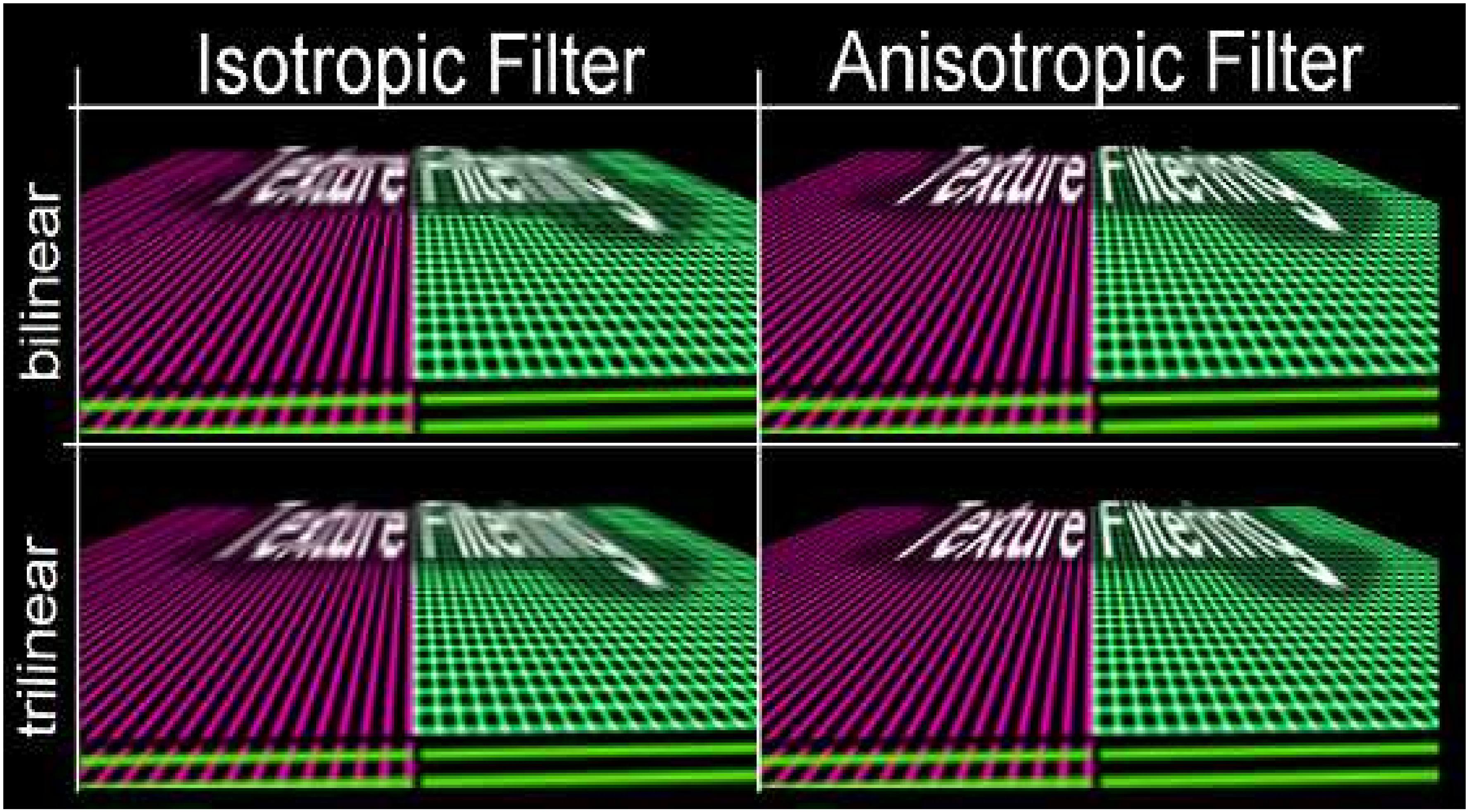
Texture space

Anisotropic Filtering: Footprint Assembly



Anti-Aliasing: Anisotropic Filtering

- Example



Texture Anti-aliasing

- Basically, everything done in hardware
- `gluBuild2DMipmaps()` generates MIPmaps
- Set parameters in `glTexParameter()`
  - `GL_TEXTURE_MAG_FILTER: GL_NEAREST, GL_LINEAR, …`
  - `GL_TEXTURE_MIN_FILTER: GL_LINEAR_MIPMAP_NEAREST`
- Anisotropic filtering is an extension:
  - `GL_EXT_texture_filter_anisotropic`
  - Number of samples can be varied (4x,8x,16x)
    - Vendor specific support and extensions

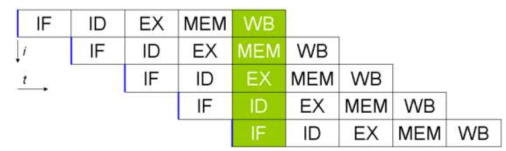
Stream Computing and GPGPU

Types of Parallelism

Bit-Level Parallelism (70s and 80s)

- Doubling the word size 4, 8, 16, 32-bit (64-bit ~2003)

Instruction-Level Parallelism (mid 80s-90s)



- Instructions are split into stages → multi stage pipeline
- Superscalar execution, …

Data Parallelism

- Multiple processors execute the same instructions on different parts of the data

Task Parallelism

- Multiple processors execute instructions independently

From GPU to GPGPU

1990s Fixed function graphics-pipeline used for more general computations in academia (e.g., rasterization, z-buffer)

2001 Shaders changed the API to access graphics cards

2004 Brook for GPUs changed the terminology

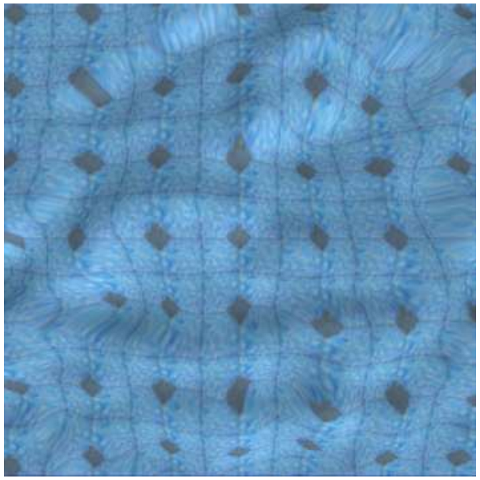
Since then:

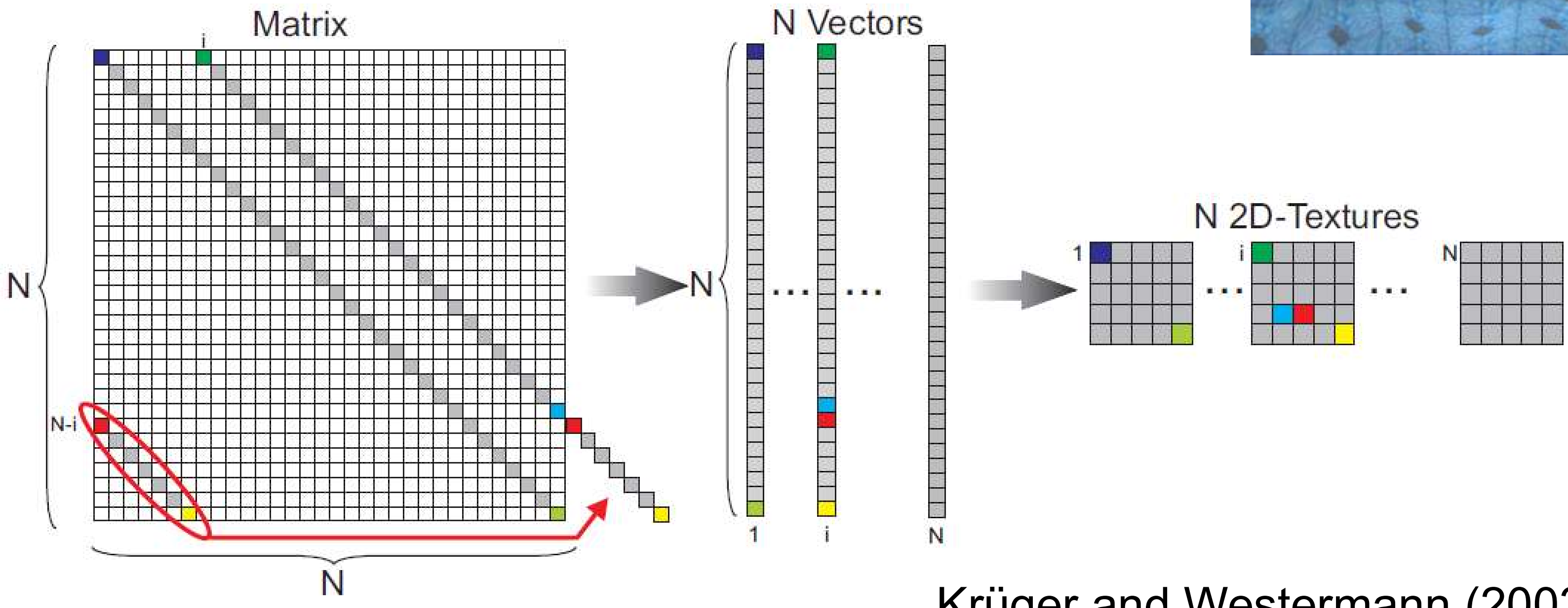
- ATI’s Stream SDK (originally based on Brook)
- NVIDIA’s CUDA (started by Brook developers)
- OpenCL (platform independent)
- GLSL Compute Shaders (platform independent)
- Vulkan Compute Shaders (platform independent)

Early GPGPU: Linear Algebra Operators

Vector and matrix representation and operators

- Early approach based on graphics primitives
- Now CUDA makes this much easier
- Linear systems solvers





Krüger and Westermann (2003)

Stream Programming Abstraction

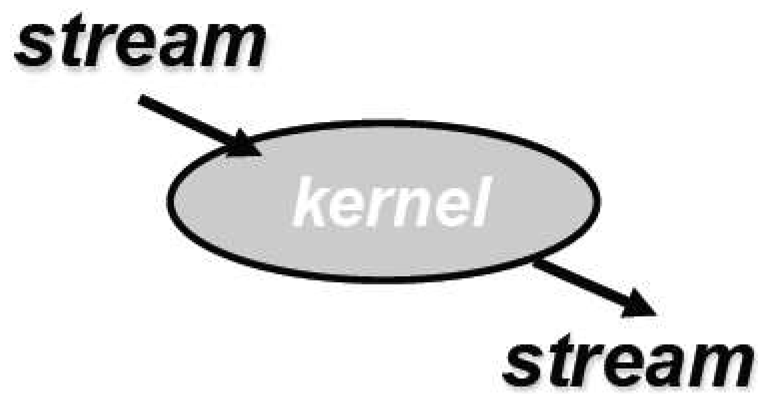
Goal: SW programming model that matches data parallelism

Streams

- Collection of data records
- All data is expressed in streams

Kernels

- Inputs/outputs are streams
- Perform computation on streams
  (each data record is processes independently)
- Can be chained together



Courtesy John Owens

Why Streams?

- Exposing parallelism
  - Data parallelism
  - Task parallelism

```
for(i = 0; i<size; i++)
{
    a[i] = 2*b[i];
}
```

```
for(each a, b)
{
    a = 2*b;
}
```

```
for(i = 0; i<size; i++)
{
    a[i] = a[i+1]*2;
}
```

```
for(each a)
{
    ???
}
```

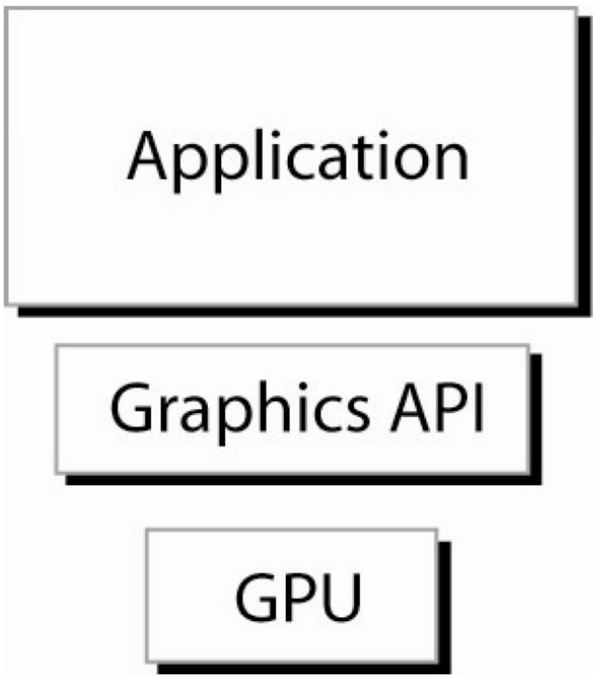
- Multiple stream elements can be processed in parallel
- Multiple tasks can be processed in parallel
- Predictable memory access pattern
- Optimize for throughput of all elements, not latency of one
- Processing many elements at once allows latency hiding

Brook for GPUs: Stream Computing on Graphics Hardware

Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan

Computer Science Department
Stanford University

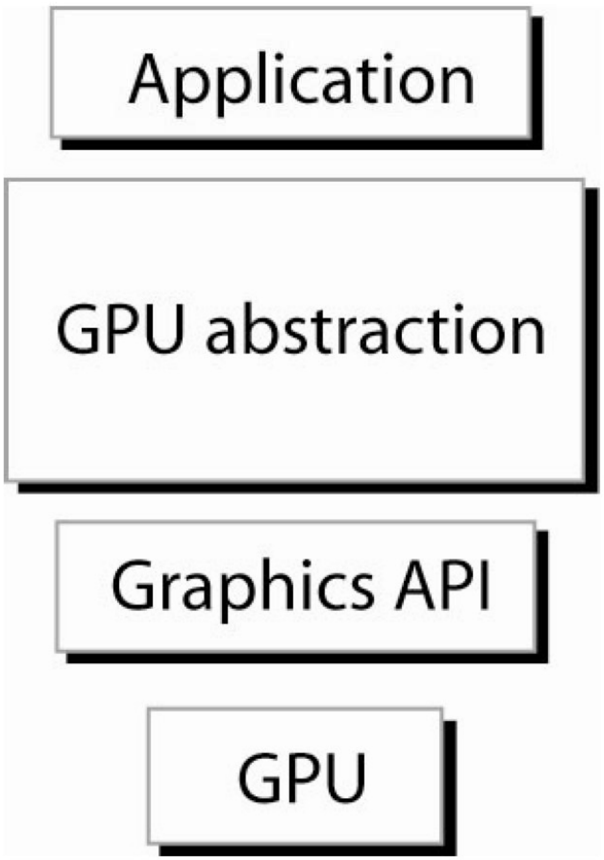
domain specific solutions



map directly to graphics primitives

requires extensive knowledge of GPU programming

building an abstraction



general GPU computing question

- can we simplify GPU programming?
- what is the correct abstraction for GPU-based computing?
- what is the scope of problems that can be implemented efficiently on the GPU?

contributions

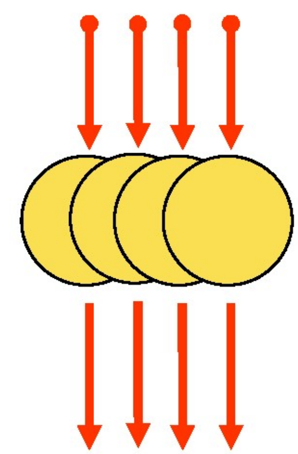
- Brook stream programming environment for GPU-based computing
  - language, compiler, and runtime system
- virtualizing or extending GPU resources
- analysis of when GPUs outperform CPUs

GPU programming model

each fragment shaded independently

– no dependencies between fragments
  - temporary registers are zeroed
  - no static variables
  - no read-modify-write textures

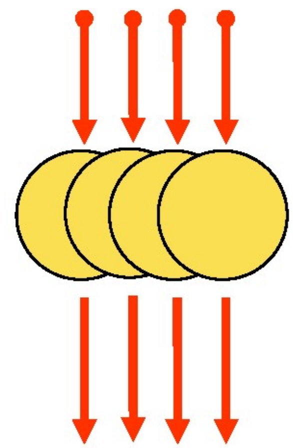
– multiple “pixel pipes”



GPU = data parallel

each fragment shaded independently

- no dependencies between fragments
  - temporary registers are zeroed
  - no static variables
  - no read-modify-write textures
- multiple “pixel pipes”



data parallelism

- support ALU heavy architectures
- hide memory latency

[Torborg and Kajiya 96, Anderson et al. 97, Igehy et al. 98]

Brook language

stream programming model

- enforce data parallel computing
  - streams
- encourage arithmetic intensity
  - kernels

design goals

- general purpose computing
  - GPU = general streaming-coprocessor
- GPU-based computing for the masses
  - no graphics experience required
  - eliminating annoying GPU limitations
- performance
- platform independent
  - ATI & NVIDIA
  - DirectX & OpenGL
  - Windows & Linux

Brook language

C with streams

- streams
  - collection of records requiring similar computation
    - particle positions, voxels, FEM cell, …

```
Ray r<200>;
float3 velocityfield<100,100,100>;
```

  - data parallelism
    - provides data to operate on in parallel

kernels

- kernels
  - functions applied to streams
    - similar to for_all construct

```
kernel void foo (float a<>, float b<>,
                 out float result<>) {
  result = a + b;
}

float a<100>;
float b<100>;
float c<100>;

foo(a,b,c);
```

```
for (i=0; i<100; i++)
      c[i] = a[i]+b[i];
```

kernels

- kernels arguments
  - input/output streams

```
kernel void foo (float a<>,
                 float b<>,
                 out float result<>) {
   result = a + b;
}
```

kernels

- kernels arguments
  - input/output streams
  - gather streams

```
kernel void foo (..., float array[] ) {
      a = array[i];
}
```

kernels

- kernels arguments
  - input/output streams
  - gather streams
  - iterator streams

```
kernel void foo (..., iter float n<> ) {
        a = n + b;
}
```

kernels

- kernels arguments
  - input/output streams
  - gather streams
  - iterator streams
  - constant parameters

```
kernel void foo (..., float c ) {
      a = c + b;
}
```

Brook language

kernels

- Ray Triangle Intersection

```
kernel void krnIntersectTriangle(Ray ray<>, Triangle tris[],
                                 RayState oldraystate<>,
                                 GridTrilist trilist[],
                                 out Hit candidatehit<>) {
  float idx, det, inv_det;
  float3 edge1, edge2, pvec, tvec, qvec;
  if(oldraystate.state.y > 0) {
    idx = trilist[oldraystate.state.w].trinum;
    edge1 = tris[idx].v1 - tris[idx].v0;
    edge2 = tris[idx].v2 - tris[idx].v0;
    pvec = cross(ray.d, edge2);
    det = dot(edge1, pvec);
    inv_det = 1.0f/det;
    tvec = ray.o - tris[idx].v0;
    candidatehit.data.y = dot( tvec, pvec ) * inv_det;
    qvec = cross( tvec, edge1 );
    candidatehit.data.z = dot( ray.d, qvec ) * inv_det;
    candidatehit.data.x = dot( edge2, qvec ) * inv_det;
    candidatehit.data.w = idx;
  } else {
    candidatehit.data = float4(0,0,0,-1);
  }
}
```

reductions

- reductions
  - compute single value from a stream

```
reduce void sum (float a<>,
                 reduce float r<>)
  r += a;
}
```

reductions

- reductions
  - compute single value from a stream

```
reduce void sum (float a<>,
                 reduce float r<>)
  r += a;
}
```

```
float a<100>;
float r;

sum(a,r);
```

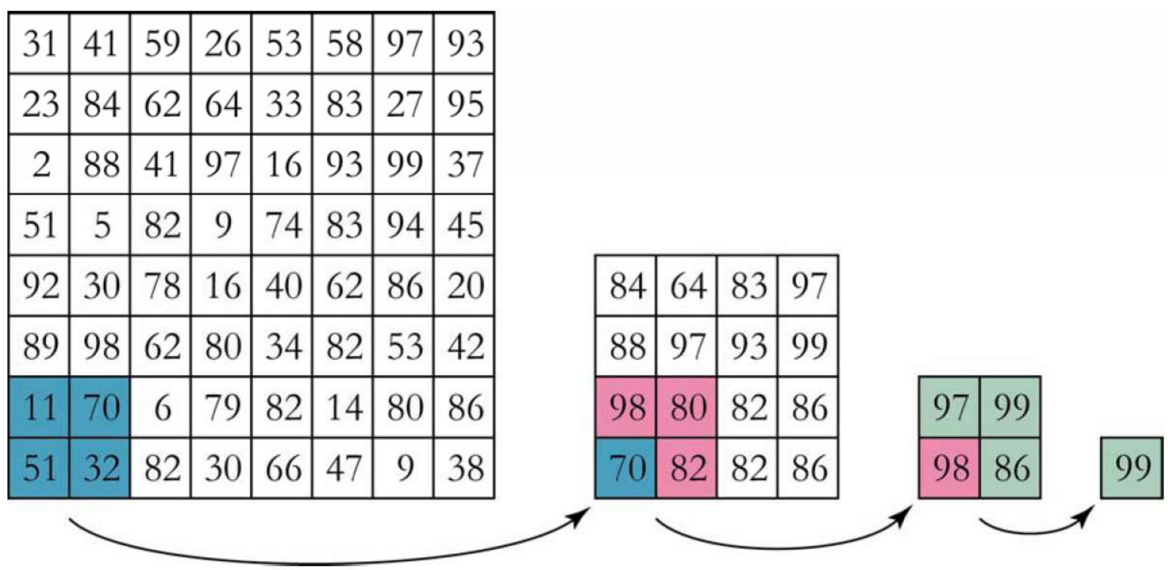
```
r = a[0];
for (int i=1; i<100; i++)
  r += a[i];
```

reductions

- reductions
  - associative operations only

    `(a+b)+c = a+(b+c)`

    - sum, multiply, max, min, OR, AND, XOR
    - matrix multiply
  - permits parallel execution



Brook language

reductions

- multi-dimension reductions
  - stream "shape" differences resolved by reduce function

reductions

- multi-dimension reductions
  - stream “shape” differences resolved by reduce function

```
reduce void sum (float a<>,
                 reduce float r<>)
  r += a;
}


float a<20>;
float r<5>;

sum(a,r);
```

Brook language

reductions

- multi-dimension reductions
  - stream “shape” differences resolved by reduce function

```
reduce void sum (float a<>,
                 reduce float r<>)
  r += a;
}


float a<20>;
float r<5>;

sum(a,r);
```

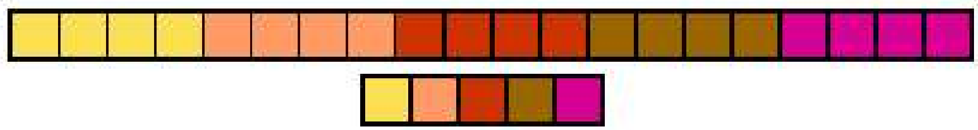
```
for (int i=0; i<5; i++)
  r[i] = a[i*4];
  for (int j=1; j<4; j++)
    r[i] += a[i*4 + j];
```

reductions

- multi-dimension reductions
  - stream "shape" differences resolved by reduce function

```
reduce void sum (float a<>,
                 reduce float r<>)
  r += a;
}
```

```
float a<20>;
float r<5>;
```



```
sum(a,r);
```

```
for (int i=0; i<5; i++)
  r[i] = a[i*4];
  for (int j=1; j<4; j++)
    r[i] += a[i*4 + j];
```

Brook language

stream repeat & stride

- kernel arguments of different shape
  - resolved by repeat and stride

stream repeat & stride

- kernel arguments of different shape
  - resolved by repeat and stride

```
kernel void foo (float a<>, float b<>,
                 out float result<>);

float a<20>;
float b<5>;
float c<10>;

foo(a,b,c);
```

stream repeat & stride

- kernel arguments of different shape
  - resolved by repeat and stride

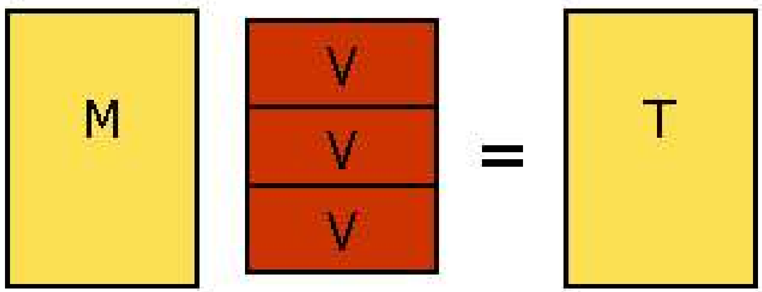
```
kernel void foo (float a<>, float b<>,
                 out float result<>);

float a<20>;
float b<5>;
float c<10>;

foo(a,b,c);
```

```
foo(a[0],  b[0], c[0])
foo(a[2],  b[0], c[1])
foo(a[4],  b[1], c[2])
foo(a[6],  b[1], c[3])
foo(a[8],  b[2], c[4])
foo(a[10], b[2], c[5])
foo(a[12], b[3], c[6])
foo(a[14], b[3], c[7])
foo(a[16], b[4], c[8])
foo(a[18], b[4], c[9])
```

Brook language

matrix vector multiply

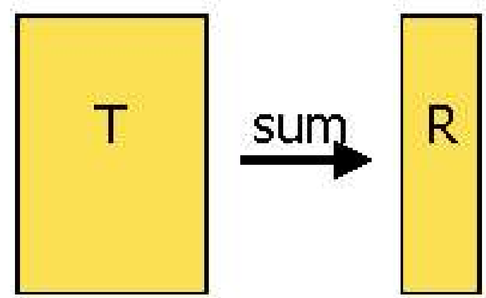
```
kernel void mul (float a<>, float b<>,
                 out float result<>) {
   result = a*b;
}

reduce void sum (float a<>,
                 reduce float result<>) {
   result += a;
}


float matrix<20,10>;
float vector<1, 10>;
float tempmv<20,10>;
float result<20, 1>;

mul(matrix,vector,tempmv);
sum(tempmv,result);
```



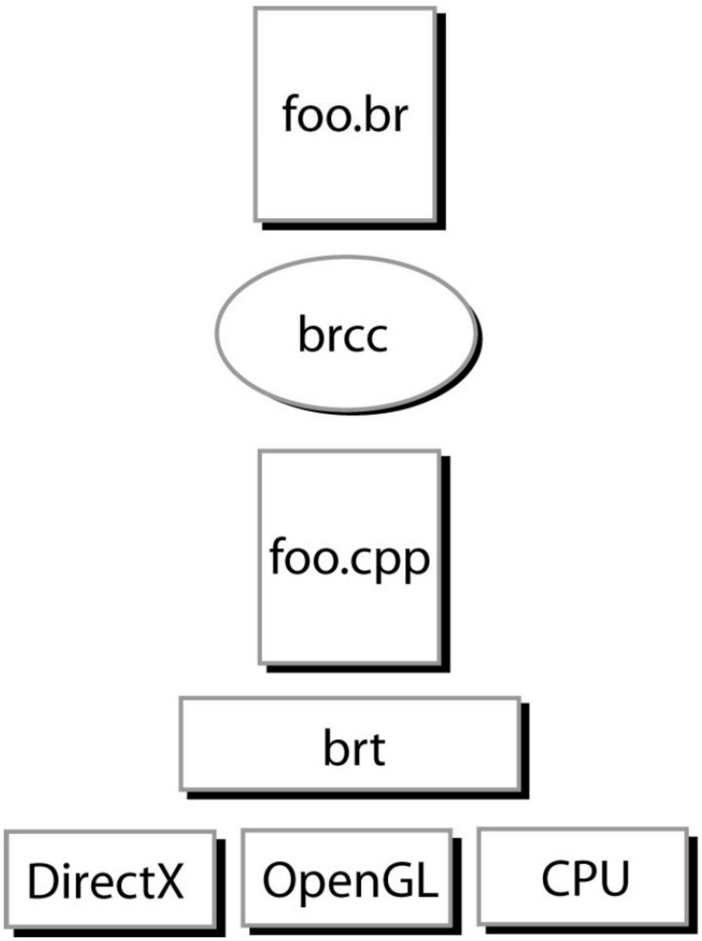
Brook language

matrix vector multiply

```
kernel void mul (float a<>, float b<>,
                 out float result<>) {
  result = a*b;
}

reduce void sum (float a<>,
                 reduce float result<>) {
  result += a;
}


float matrix<20,10>;
float vector<1, 10>;
float tempmv<20,10>;
float result<20, 1>;

mul(matrix,vector,tempmv);
sum(tempmv,result);
```



system outline



brcc

source to source compiler
- generate CG & HLSL code
- CGC and FXC for shader assembly
- virtualization

brt

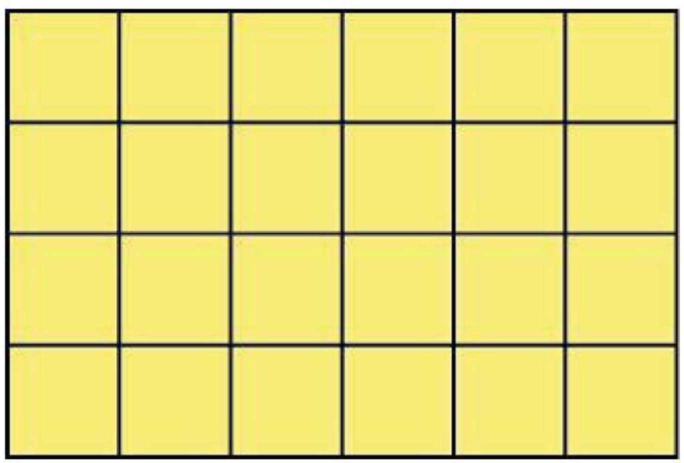
Brook run-time library
- stream texture management
- kernel shader execution

eliminating GPU limitations

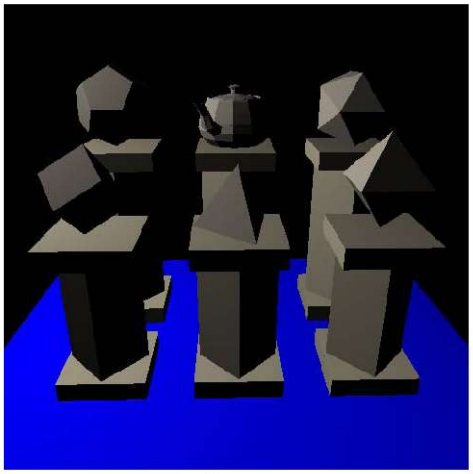
treating texture as memory

- limited texture size and dimension
- compiler inserts address translation code

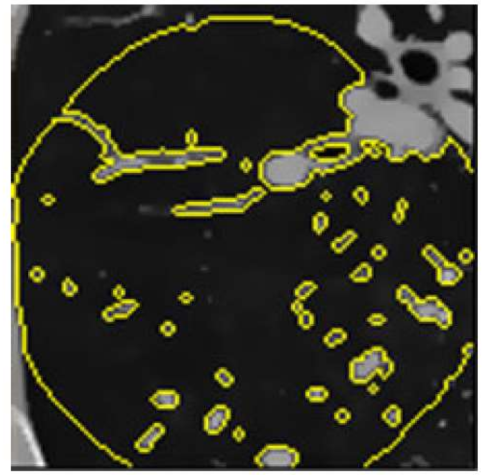
```
float matrix<8096,10,30,5>;
```



applications

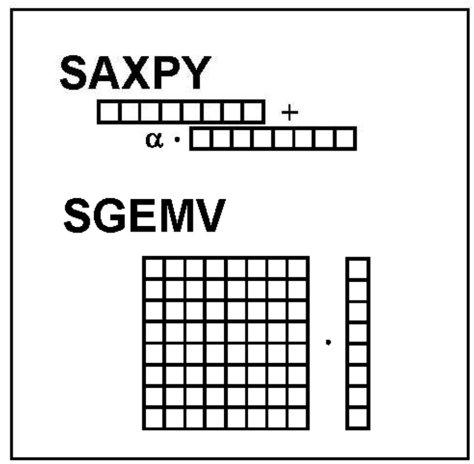


ray-tracer



segmentation

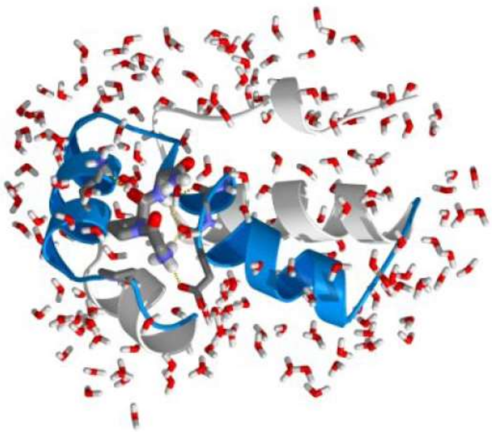
fft edge detect



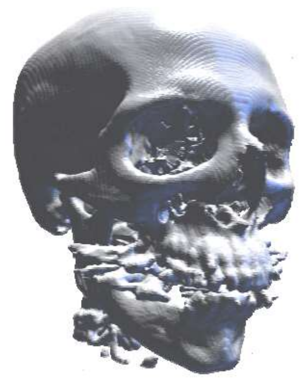
linear algebra

summary

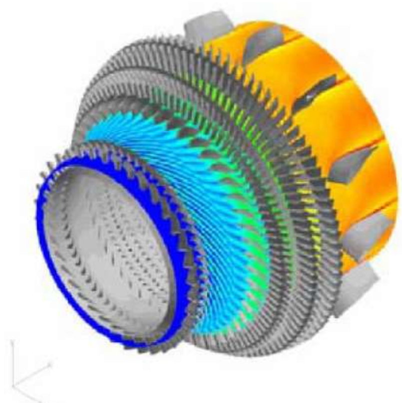
GPU-based computing for the masses



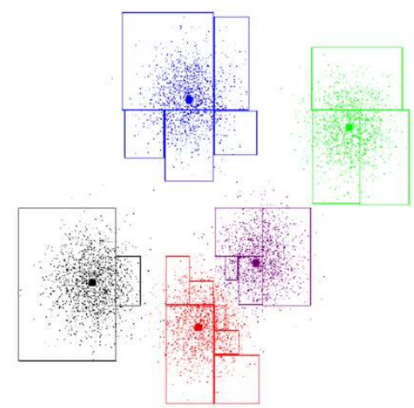
bioinfomatics



rendering



simulation



statistics

Thank you.

- John Owens
- Ian Buck et al.
- AMD