CPSC/ECE 4780/6780

General-Purpose Computation on Graphical Processing Units (GPGPU)

Lecture 6: CUDA Memories by Examples

Recaps of Last Lecture

- Compute to global memory access (CGMA) ratio
- CUDA device memory types
  - Global memory
  - Shared memory
  - Constant memory
  - Registers
  - Texture memory
- Scope and lifetime of CUDA variables
- Reducing global memory access enhances performance
- Memory coalescing
- Shared memory banks and bank conflicts

CUDA Memories by Examples

- Global memory example
  - Julia Set
- Shared memory example
  - Shared memory bitmap
- Constant memory example
  - Ray tracing
- Texture memory example
  - Heat transfer simulation

Julia Set

- Julia Set is the boundary of a certain class of functions over complex numbers
- Julia set evaluates a simple iterative equation for points in the complex plane
  - A point is not in the set if the process of iterating the equation diverges (grows toward infinity) for that point
  - If the values taken by the equation remain bounded, the point is in the set
- Iterative equation

$$Z_{n+1} = Z_n^2 + C$$

involves squaring the current value and adding a constant to get the next value of the equation

CPU Julia Set – Main Function

- Create the appropriate size bitmap image using a utility library provided by NVIDIA
- Pass a pointer to the bitmap data to the kernel function
- Display the bitmap and exit

```
int main( void ) {
    CPUBitmap bitmap( DIM, DIM );
    unsigned char *ptr = bitmap.get_ptr();

    kernel( ptr );

    bitmap.display_and_exit();
}
```

CPU Julia Set – Computation Kernel

- Iterate through all points we care to render
- Call julia() on each point to determine its membership in the Julia Set
  - Return 1 if the point is in the set
  - Return 0 otherwise

```
void kernel( unsigned char *ptr ){
    for (int y=0; y<DIM; y++) {
        for (int x=0; x<DIM; x++) {
            int offset = x + y * DIM;

            int juliaValue = julia( x, y );
            ptr[offset*4 + 0] = 255 * juliaValue;
            ptr[offset*4 + 1] = 0;
            ptr[offset*4 + 2] = 0;
            ptr[offset*4 + 3] = 255;
        }
    }
}
```

CPU Julia Set – Julia()

- Translate pixel coordinate to a coordinate in complex space
- Set the arbitrary complex-valued constant C
- Iterate 200 times and determine whether the point is in or out of the Julia Set

Center the complex plane at the image center

```
int julia( int x, int y ) {
    const float scale = 1.5;
    float jx = scale * (float)(DIM/2 - x)/(DIM/2);
    float jy = scale * (float)(DIM/2 - y)/(DIM/2);

    cuComplex c(-0.8, 0.156);
    cuComplex a(jx, jy);

    int i = 0;
    for (i=0; i<200; i++) {
        a = a * a + c;
        if (a.magnitude2() > 1000)
            return 0;
    }

    return 1;
}
```

Zoom in or out

Ensure the image spans the range of (-1.0, 1.0)

CPU Julia Set – cuComplex Struct

- Define a generic structure to store complex numbers

```
struct cuComplex {
    float   r;
    float   i;
    cuComplex( float a, float b ) : r(a), i(b)  {}
    float magnitude2( void ) { return r * r + i * i; }
    cuComplex operator*(const cuComplex& a) {
        return cuComplex(r*a.r - i*a.i, i*a.r + r*a.i);
    }
    cuComplex operator+(const cuComplex& a) {
        return cuComplex(r+a.r, i+a.i);
    }
};
```

GPU Julia Set – Main Function

```
int main( void ) {
    CPUBitmap bitmap( DIM, DIM );
    unsigned char *ptr = bitmap.get_ptr();

    kernel( ptr );

    bitmap.display_and_exit();
}
```

CPU version counterpart

- Declare a pointer dev_bitmap to hold a copy of the data on the device
- Allocate memory using cudaMalloc
- Specify a two-dimensional grid of blocks for kernel computation
- Copy the results back to the host
- Display the bitmap and exit

```
int main( void ) {
    CPUBitmap bitmap( DIM, DIM, &data );
    unsigned char    *dev_bitmap;

    HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap, bitmap.
        image_size() ) );

    dim3    grid(DIM,DIM);
    kernel<<<grid,1>>>( dev_bitmap );

    HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(), dev_bitmap,
                              bitmap.image_size(),
                              cudaMemcpyDeviceToHost ) );
    HANDLE_ERROR( cudaFree( dev_bitmap ) );
    bitmap.display_and_exit();
}
```

GPU Julia Set – Computation Kernel

- Declare kernel() as a __global__ function so it runs on the device but can be called from the host
- Calculate pixel index using blockIdx instead of nested for() loops
- Calculate a linear offset using gridDim
- Call Julia() on each to determine membership in the Julia Set

```
__global__ void kernel( unsigned char *ptr ) {
    // map from blockIdx to pixel position
    int x = blockIdx.x;
    int y = blockIdx.y;
    int offset = x + y * gridDim.x;

    // now calculate the value at that position
    int juliaValue = julia( x, y );
    ptr[offset*4 + 0] = 255 * juliaValue;
    ptr[offset*4 + 1] = 0;
    ptr[offset*4 + 2] = 0;
    ptr[offset*4 + 3] = 255;
}
```

```
void kernel( unsigned char *ptr ){
    for (int y=0; y<DIM; y++) {
        for (int x=0; x<DIM; x++) {
            int offset = x + y * DIM;

            int juliaValue = julia( x, y );
            ptr[offset*4 + 0] = 255 * juliaValue;
            ptr[offset*4 + 1] = 0;
            ptr[offset*4 + 2] = 0;
            ptr[offset*4 + 3] = 255;
        }
    }
}
```

CPU version counterpart

GPU Julia Set – Julia()

```
__device__ int julia( int x, int y ) {
    const float scale = 1.5;
    float jx = scale * (float)(DIM/2 - x)/(DIM/2);
    float jy = scale * (float)(DIM/2 - y)/(DIM/2);

    cuComplex c(-0.8, 0.156);
    cuComplex a(jx, jy);

    int i = 0;
    for (i=0; i<200; i++) {
        a = a * a + c;
        if (a.magnitude2() > 1000)
            return 0;
    }

    return 1;
}
```

```
int julia( int x, int y ) {
    const float scale = 1.5;
    float jx = scale * (float)(DIM/2 - x)/(DIM/2);
    float jy = scale * (float)(DIM/2 - y)/(DIM/2);

    cuComplex c(-0.8, 0.156);
    cuComplex a(jx, jy);

    int i = 0;
    for (i=0; i<200; i++) {
        a = a * a + c;
        if (a.magnitude2() > 1000)
            return 0;
    }

    return 1;
}
```

CPU version counterpart

GPU Julia Set – cuComplex Struct

```
struct cuComplex {
    float   r;
    float   i;
    __host__ __device__ cuComplex( float a, float b ) : r(a), i(b)
        {}
    __device__ float magnitude2( void ) {
        return r * r + i * i;
    }
    __device__ cuComplex operator*(const cuComplex& a) {
        return cuComplex(r*a.r - i*a.i, i*a.r + r*a.i);
    }
    __device__ cuComplex operator+(const cuComplex& a) {
        return cuComplex(r+a.r, i+a.i);
    }
};
```

```
struct cuComplex {
    float   r;
    float   i;
    cuComplex( float a, float b ) : r(a), i(b)  {}
    float magnitude2( void ) { return r * r + i * i; }
    cuComplex operator*(const cuComplex& a) {
        return cuComplex(r*a.r - i*a.i, i*a.r + r*a.i);
    }
    cuComplex operator+(const cuComplex& a) {
        return cuComplex(r+a.r, i+a.i);
    }
};
```

CPU version counterpart

GPU Julia Set – Compilation

- nvcc -lglut -lGL -lGLU juliaSet.cu

Shared Memory Bitmap – Main Function

```
struct DataBlock {
    unsigned char   *dev_bitmap;
};

int main( void ) {
    DataBlock   data;
    CPUBitmap bitmap( DIM, DIM, &data );
    unsigned char    *dev_bitmap;

    HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap,
                              bitmap.image_size() ) );

    data.dev_bitmap = dev_bitmap;

    dim3    grids(DIM/16,DIM/16);
    dim3    threads(16,16);
    kernel<<<grids,threads>>>( dev_bitmap );

    HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(), dev_bitmap,
                              bitmap.image_size(),
                              cudaMemcpyDeviceToHost ) );

    HANDLE_ERROR( cudaFree( dev_bitmap ) );

    bitmap.display_and_exit();
}
```

Launch multiple threads per block

Shared Memory Bitmap – Kernel Function

```
__global__ void kernel( unsigned char *ptr ) {
    // map from threadIdx/BlockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    __shared__ float shared[16][16];

    // now calculate the value at that position
    const float period = 128.0f;

    shared[threadIdx.x][threadIdx.y] =
            255 * (sinf(x*2.0f*PI/ period) + 1.0f) *
                  (sinf(y*2.0f*PI/ period) + 1.0f) / 4.0f;

    __syncthreads();

    ptr[offset*4 + 0] = 0;
    ptr[offset*4 + 1] = shared[15-threadIdx.x][15-threadIdx.y];
    ptr[offset*4 + 2] = 0;
    ptr[offset*4 + 3] = 255;
}
```

Compute x and y location in the output image

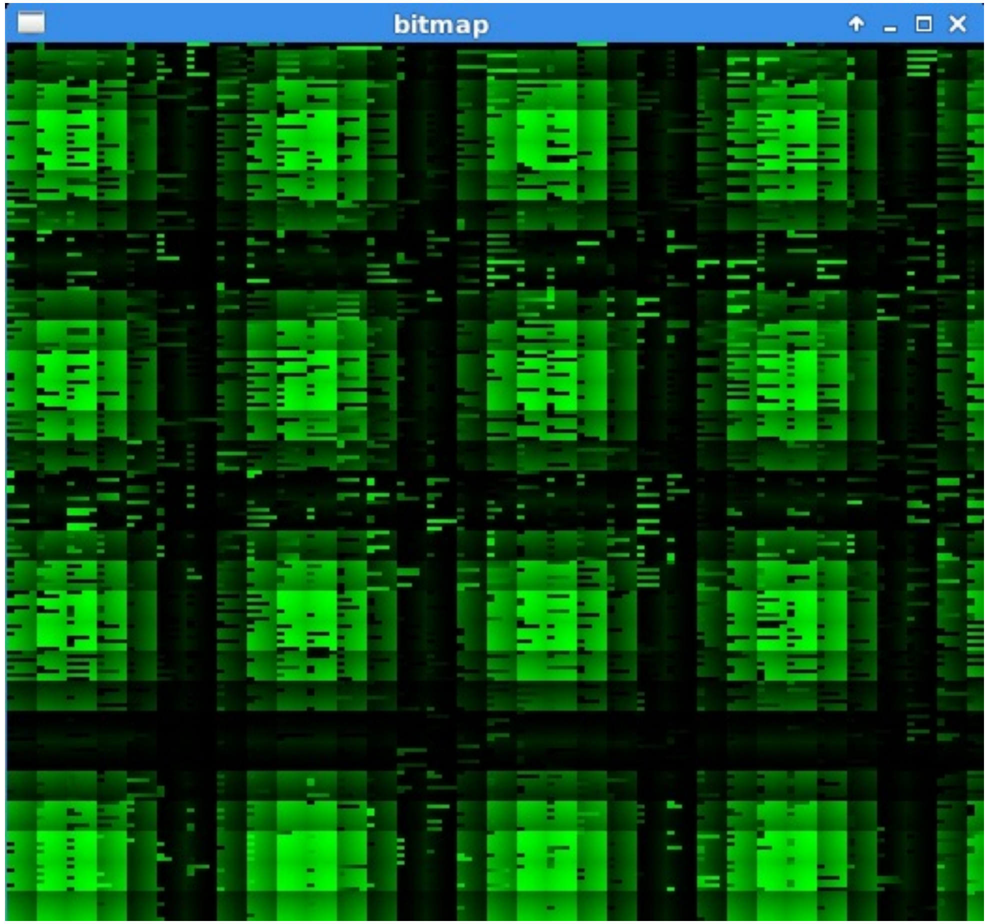
Use a shared memory buffer to cache the computation

Each thread computes a value to be stored into the buffer

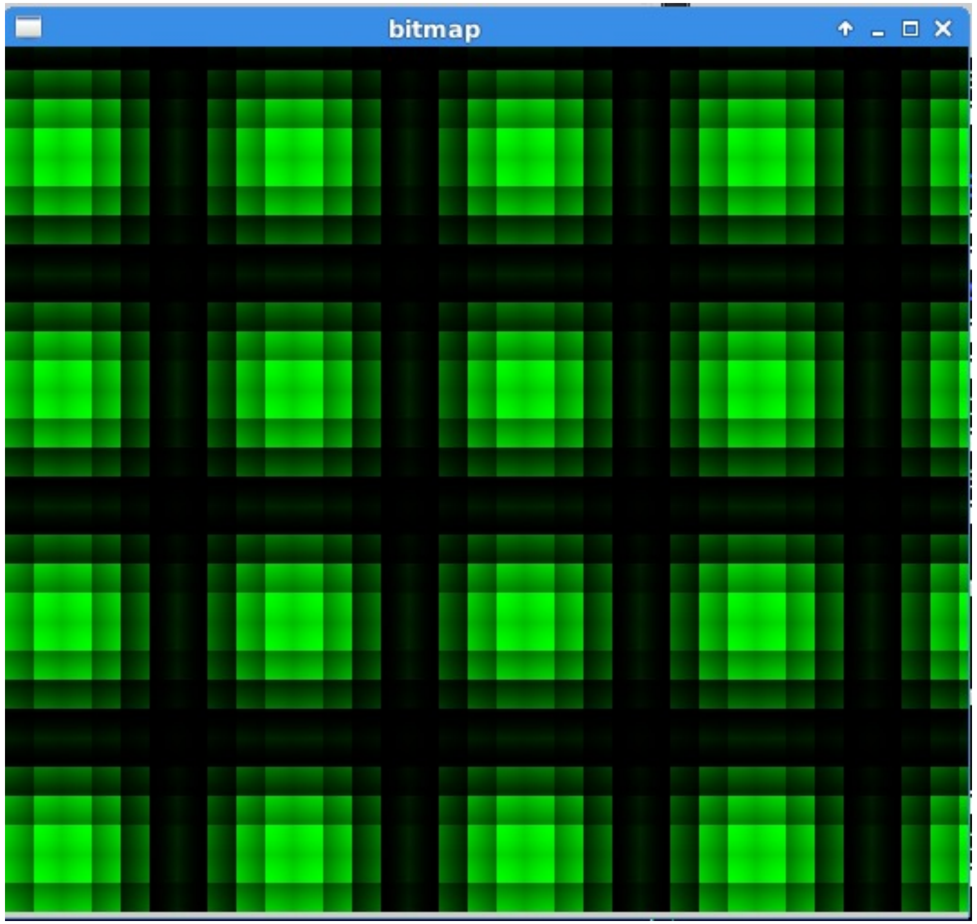
To make sure all the values are written before the subsequent read from it

Store the values back to the pixel, reversing the order of x and y

Shared Memory Bitmap – Comparison on Correctness



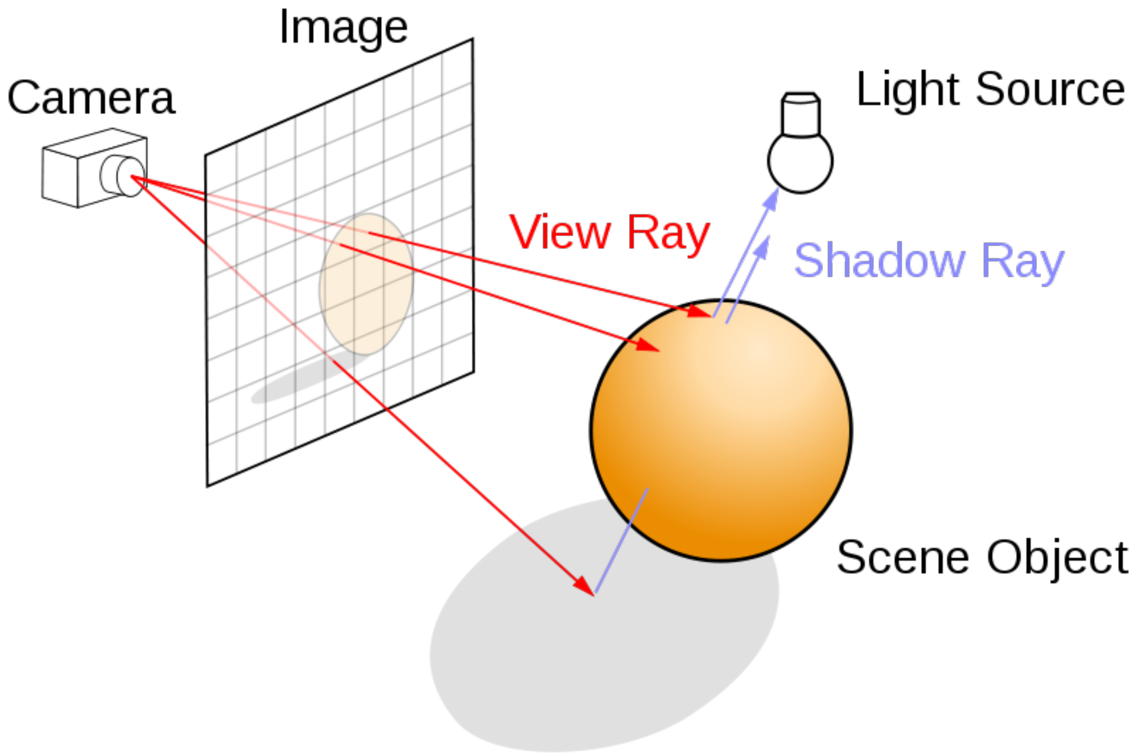
No __syncthreads()



with __syncthreads()

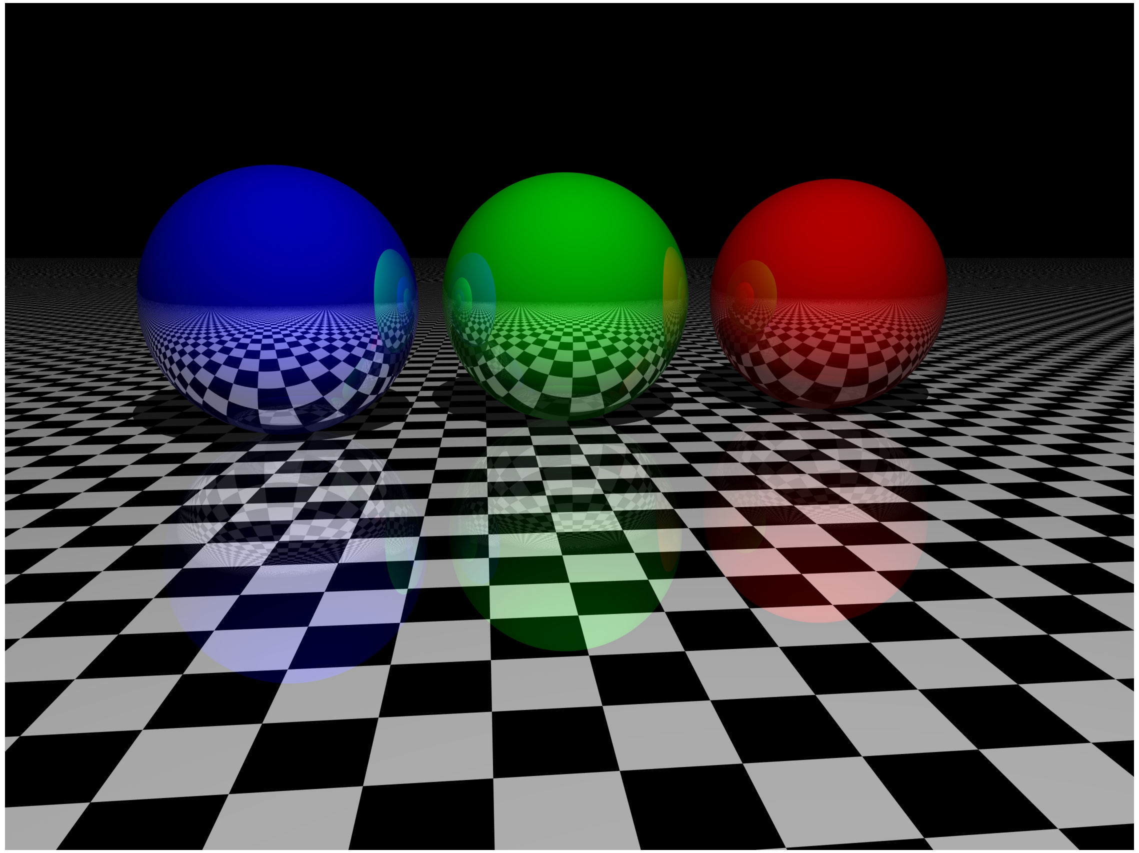
Ray Tracing

- Ray tracing is a technique for generating an image by tracing the path of light as pixels in an image plane and simulating the effects of its encounters with virtual objects



How Ray Tracing Works

- 1. For each pixel create a ray
- 2. Follow the ray into the scene and find the closest object that is intersected and assign the color of the object to the pixel
- 3. If the object is made of glass, mirror or something similar, find the reflected and/or refracted ray and trace on
- 4. Else, compute the color of the pixel based on light sources and material information



A Basic Ray Tracer

- Restrictions
  - Support scenes of spheres only
  - The camera is restricted to the z-axis, facing the origin
  - No support of any lighting of the scene
  - Each sphere will be assigned a color and shaded with precomputed function if they are visible
- What will it do?
  - Fire a ray from each pixel and keep track of which rays hit which spheres
  - Track the depth of each of these hits to find the closest sphere

Ray Tracing on GPU – Sphere Struct

```
#define INF     2e10f

struct Sphere {
    float   r,b,g;
    float   radius;
    float   x,y,z;
    __device__ float hit( float ox, float oy, float *n ) {
        float dx = ox - x;
        float dy = oy - y;
        if (dx*dx + dy*dy < radius*radius) {
            float dz = sqrtf( radius*radius - dx*dx - dy*dy );
            *n = dz / sqrtf( radius * radius );
            return dz + z;
        }
        return -INF;
    }
};
```

- The "hit" method computes whether the ray intersects the sphere

Ray Tracing on GPU – Main Function

```
int main( void ) {
    CPUBitmap bitmap( DIM, DIM );
    unsigned char   *dev_bitmap;
    Sphere          *s;

    // allocate memory on the GPU for the output bitmap
    HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap,
                              bitmap.image_size() ) );
    // allocate memory for the Sphere dataset
    HANDLE_ERROR( cudaMalloc( (void**)&s,
                              sizeof(Sphere) * SPHERES ) );

    // allocate temp memory, initialize it, copy to
    // memory on the GPU, then free our temp memory
    Sphere *temp_s = (Sphere*)malloc( sizeof(Sphere) * SPHERES )
    for (int i=0; i<SPHERES; i++) {
        temp_s[i].r = rnd( 1.0f );
        temp_s[i].g = rnd( 1.0f );
        temp_s[i].b = rnd( 1.0f );
        temp_s[i].x = rnd( 1000.0f ) - 500;
        temp_s[i].y = rnd( 1000.0f ) - 500;
        temp_s[i].z = rnd( 1000.0f ) - 500;
        temp_s[i].radius = rnd( 100.0f ) + 20;
    }
    HANDLE_ERROR( cudaMemcpy( s, temp_s,
                                sizeof(Sphere) * SPHERES,
                                cudaMemcpyHostToDevice ) );
    free( temp_s );

    // generate a bitmap from our sphere data
    dim3    grids(DIM/16,DIM/16);
    dim3    threads(16,16);
    kernel<<<grids,threads>>>( s, dev_bitmap );

    // copy our bitmap back from the GPU for display
    HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(), dev_bitmap,
                              bitmap.image_size(),
                              cudaMemcpyDeviceToHost ) );

    HANDLE_ERROR( cudaFree( dev_bitmap ) );
    HANDLE_ERROR( cudaFree( s ) );

    // display
    bitmap.display_and_exit();
}
```

Ray Tracing on GPU – Kernel Function

```
__global__ void kernel( Sphere *s, unsigned char *ptr ) {
    // map from threadIdx/BlockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;
    float   ox = (x - DIM/2);
    float   oy = (y - DIM/2);

    float   r=0, g=0, b=0;
    float   maxz = -INF;
    for(int i=0; i<SPHERES; i++) {
        float   n;
        float   t = s[i].hit( ox, oy, &n );
        if (t > maxz) {
            float fscale = n;
            r = s[i].r * fscale;
            g = s[i].g * fscale;
            b = s[i].b * fscale;
            maxz = t;
        }
    }

    ptr[offset*4 + 0] = (int)(r * 255);
    ptr[offset*4 + 1] = (int)(g * 255);
    ptr[offset*4 + 2] = (int)(b * 255);
    ptr[offset*4 + 3] = 255;
}
```

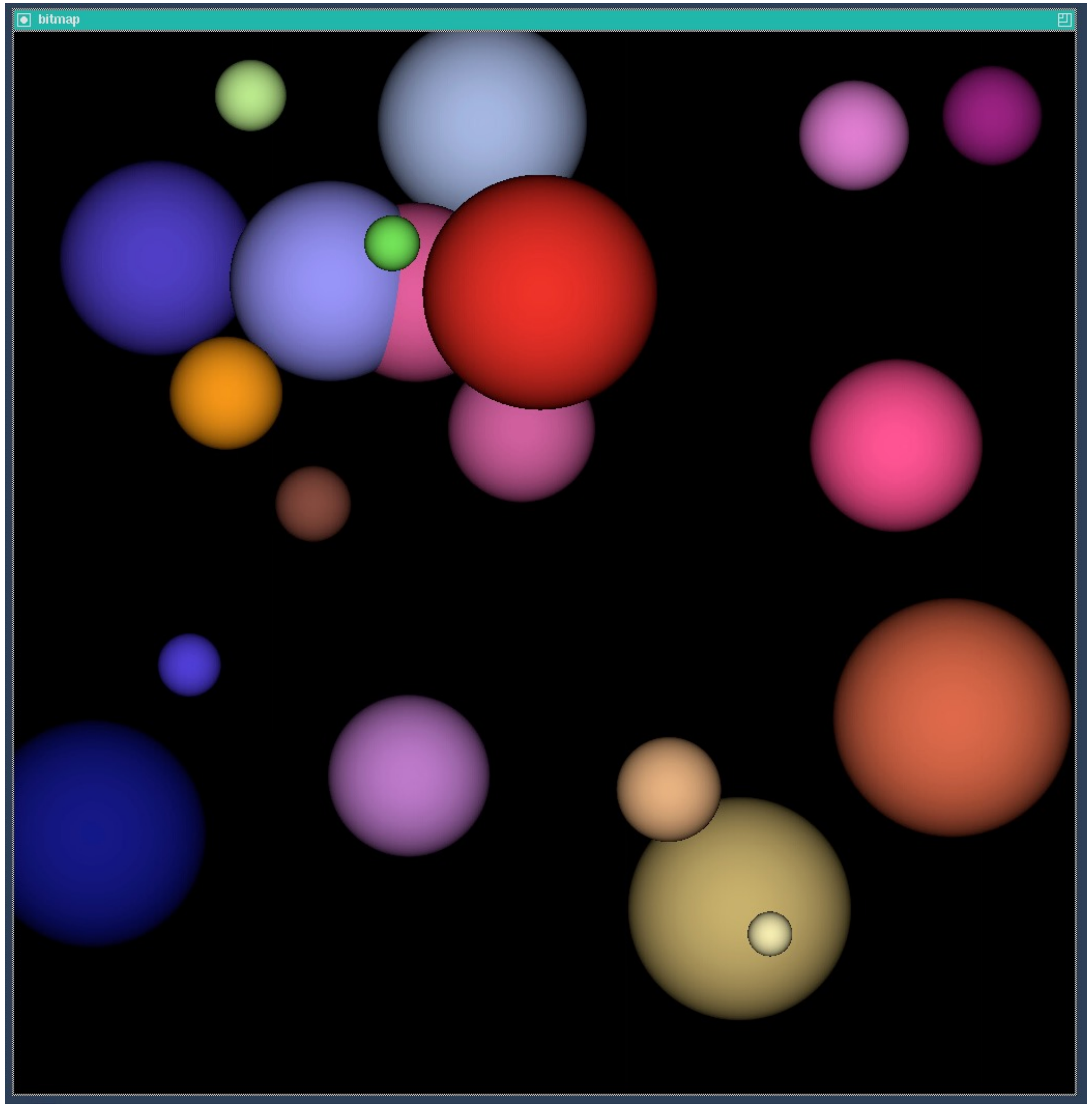
Each thread generates one pixel for the output image

Initialize the background to be black if no spheres have been hit

Iterate through the array of spheres to check each sphere for intersection

Store the current color into the output image

Ray Tracing on GPU – Screenshot



Ray Tracing on GPU with Constant Memory

- Store the input array of spheres in constant memory
  - Instead of declaring the array like

    Sphere *s; → Declare a pointer and use cudaMalloc() to allocate GPU memory for it

    We add the modifier __constant__ before it:

    __constant__ Sphere s[SPHERES]; → Statically allocate the space in constant memory
- Need to commit to a size for this array at compile-time
- Changes in main():

```
HANDLE_ERROR( cudaMemcpy( s, temp_s,
                          sizeof(Sphere) * SPHERES,
                          cudaMemcpyHostToDevice ) );
```

↓

```
HANDLE_ERROR( cudaMemcpyToSymbol( s, temp_s,
                          sizeof(Sphere) * SPHERES) );
```

Benefits of Using Constant Memory for Ray Tracing on GPU

- Receive the data in a half-warp broadcast is efficient
- Retrieve the data from the constant memory cache is fast

Measuring Performance with Events

- CUDA event API: An event in CUDA is essentially a GPU time stamp that is recorded at a user-specified point in time.
- Two steps:
  - Creating an event
  - Subsequently recording an event

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);

// do some work on the GPU

cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
```

Make sure all GPU work before the stop event has completed so it is safe to read the tie stamp recorded

- Do not use CUDA events to time mixtures of device and host code!

Measuring Ray Tracing Performance

```
// capture the start time
cudaEvent_t     start, stop;
HANDLE_ERROR( cudaEventCreate( &start ) );
HANDLE_ERROR( cudaEventCreate( &stop ) );
HANDLE_ERROR( cudaEventRecord( start, 0 ) );
```

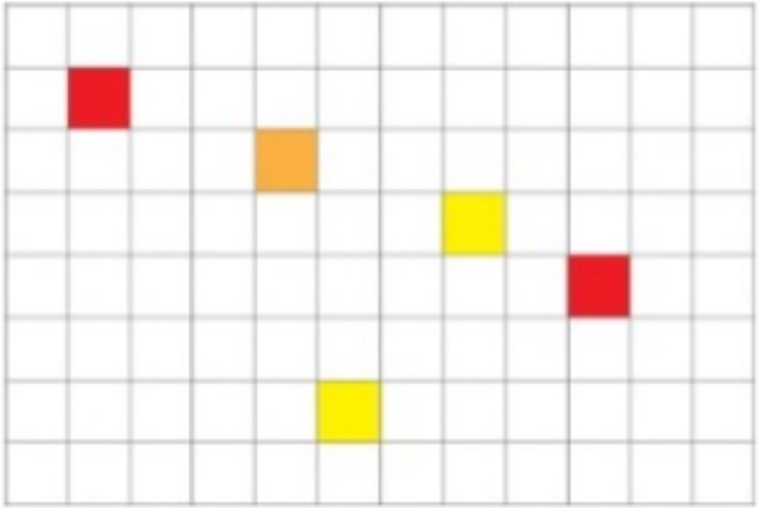
```
// get stop time, and display the timing results
HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
HANDLE_ERROR( cudaEventSynchronize( stop ) );
float   elapsedTime;
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                    start, stop ) );
printf( "Time to generate:  %3.1f ms\n", elapsedTime );

HANDLE_ERROR( cudaEventDestroy( start ) );
HANDLE_ERROR( cudaEventDestroy( stop ) );
```

Computes the elapsed time between two previously recorded events

Free the created events

Heat Transfer Simulation

- Simple heating model
  - A rectangular room divide into a grid with "heaters" of various temperature
  - Cells with heaters in them always remain a constant temperature
  - Heat "flows" between a cell and its neighbors
  - New temperature in a cell is the sum of differences between its temperature and the temperatures of its neighbor



$$T_{NEW} = T_{OLD} + \sum_{NEIGHBORS} k \cdot (T_{NEIGHBOR} - T_{OLD})$$

$$T_{NEW} = T_{OLD} + k \cdot (T_{TOP} + T_{BOTTOM} + T_{LEFT} + T_{RIGHT} - 4 \cdot T_{OLD})$$

Heat Transfer Simulation – Computing Temperature Updates

- Step 1. Given some grid of input temperatures, copy the temperature of cells with heaters to this grid, overwriting any previously computed temperatures in these cells
  - copy_const_kernel()
- Step 2. Given the input temperature grid, compute the output temperatures based on the update equation
  - Blend_kernel()
- Step 3. Swap the input and output buffers in preparation of the next time step

Heat Transfer Simulation – Step 1

```
__global__ void copy_const_kernel( float *iptr,
                                   const float *cptr ) {
    // map from threadIdx/BlockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    if (cptr[offset] != 0)  iptr[offset] = cptr[offset];
}
```

Heat Transfer Simulation – Step 2

```
__global__ void blend_kernel( float *outSrc,
                              const float *inSrc ) {
    // map from threadIdx/BlockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    int left = offset - 1;
    int right = offset + 1;
    if (x == 0)   left++;
    if (x == DIM-1) right--;

    int top = offset - DIM;
    int bottom = offset + DIM;
    if (y == 0)   top += DIM;
    if (y == DIM-1) bottom -= DIM;

    outSrc[offset] = inSrc[offset] + SPEED * ( inSrc[top] +
                     inSrc[bottom] + inSrc[left] + inSrc[right] -
                     inSrc[offset]*4);
}
```

Determine the offsets of the left, right, top, and bottom neighbors

$$\boxed{T_{NEW} = T_{OLD} + k \cdot (T_{TOP} + T_{BOTTOM} + T_{LEFT} + T_{RIGHT} - 4 \cdot T_{OLD})}$$

Adding the old temperature and the scaled differences of that temperature and the cell's neighbors' temperature

Animating the Simulation – anim_gpu()

```
void anim_gpu( DataBlock *d, int ticks ) {
    ......
    for (int i=0; i<90; i++) {
        copy_const_kernel<<<blocks,threads>>>( d->dev_inSrc,
                                               d->dev_constSrc );
        blend_kernel<<<blocks,threads>>>( d->dev_outSrc,
                                          d->dev_inSrc );
        swap( d->dev_inSrc, d->dev_outSrc );
    }
    float_to_color<<<blocks,threads>>>( d->output_bitmap,
                                        d->dev_inSrc );

    HANDLE_ERROR( cudaMemcpy( bitmap->get_ptr(),
                              d->output_bitmap,
                              bitmap->image_size(),
                              cudaMemcpyDeviceToHost ) );
    ......
}
```

Called by the animation framework on every frame

Compute a single time step of the simulation as described by Step 1 to 3

Convert the temperatures to colors

Copy the resultant image back to the CPU

Using Texture Memory for Heat Transfer Simulation – Set Up Textures

- First, declare inputs as texture references
  - texture<float> texConstSrc;
  - texture<float> texIn;
  - texture<float> texOut;
- Next, bind the references to the memory buffer using cudaBindTexture()

```
HANDLE_ERROR( cudaMalloc( (void**)&data.dev_inSrc,
                          imageSize ) );
HANDLE_ERROR( cudaMalloc( (void**)&data.dev_outSrc,
                          imageSize ) );
HANDLE_ERROR( cudaMalloc( (void**)&data.dev_constSrc,
                          imageSize ) );

HANDLE_ERROR( cudaBindTexture( NULL, texConstSrc,
                               data.dev_constSrc,
                               imageSize ) );

HANDLE_ERROR( cudaBindTexture( NULL, texIn,
                               data.dev_inSrc,
                               imageSize ) );

HANDLE_ERROR( cudaBindTexture( NULL, texOut,
                               data.dev_outSrc,
                               imageSize ) );
```

Using Texture Memory for Heat Transfer Simulation – Reading from Texture Memory

- Modify blend_kernel() to use tex1Dfetch() when reading from memory
- Texture references must be declared globally at file scope
  - No longer pass the input and output buffers as parameters to blend_kernel()
  - Pass to blend_kernel() a boolean flag dstOut that indicates which buffer to use as input and which to use as output

```
float   t, l, c, r, b;
if (dstOut) {
    t = tex1Dfetch(texIn,top);
    l = tex1Dfetch(texIn,left);
    c = tex1Dfetch(texIn,offset);
    r = tex1Dfetch(texIn,right);
    b = tex1Dfetch(texIn,bottom);

} else {
    t = tex1Dfetch(texOut,top);
    l = tex1Dfetch(texOut,left);
    c = tex1Dfetch(texOut,offset);
    r = tex1Dfetch(texOut,right);
    b = tex1Dfetch(texOut,bottom);
}
dst[offset] = c + SPEED * (t + b + r + l - 4 * c);
```

```
__global__ void blend_kernel( float *dst,
                              bool dstOut ) {
```

Using Texture Memory for Heat Transfer Simulation – Changes in copy_const_kernel()

```
__global__ void copy_const_kernel( float *iptr ) {
    // map from threadIdx/BlockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    float c = tex1Dfetch(texConstSrc,offset);
    if (c != 0)
        iptr[offset] = c;
}
```

```
__global__ void copy_const_kernel( float *iptr,
                                   const float *cptr ) {
    // map from threadIdx/BlockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    if (cptr[offset] != 0)  iptr[offset] = cptr[offset];
}
```

Non-texture version counterpart

Using Texture Memory for Heat Transfer Simulation – Changes in anim_gpu()

```
volatile bool dstOut = true;
for (int i=0; i<90; i++) {
    float   *in, *out;
    if (dstOut) {
        in  = d->dev_inSrc;
        out = d->dev_outSrc;
    } else {
        out = d->dev_inSrc;
        in  = d->dev_outSrc;
    }
    copy_const_kernel<<<blocks,threads>>>( in );
    blend_kernel<<<blocks,threads>>>( out, dstOut );
    dstOut = !dstOut;
}
float_to_color<<<blocks,threads>>>( d->output_bitmap,
                                    d->dev_inSrc );

HANDLE_ERROR( cudaMemcpy( bitmap->get_ptr(),
                          d->output_bitmap,
                          bitmap->image_size(),
                          cudaMemcpyDeviceToHost ) );
```

Using Texture Memory for Heat Transfer Simulation – Unbind Texture

```
// clean up memory allocated on the GPU
void anim_exit( DataBlock *d ) {
    cudaUnbindTexture( texIn );
    cudaUnbindTexture( texOut );
    cudaUnbindTexture( texConstSrc );
    HANDLE_ERROR( cudaFree( d->dev_inSrc ) );
    HANDLE_ERROR( cudaFree( d->dev_outSrc ) );
    HANDLE_ERROR( cudaFree( d->dev_constSrc ) );

    HANDLE_ERROR( cudaEventDestroy( d->start ) );
    HANDLE_ERROR( cudaEventDestroy( d->stop ) );
}
```

Using Two-Dimensional Texture Memory

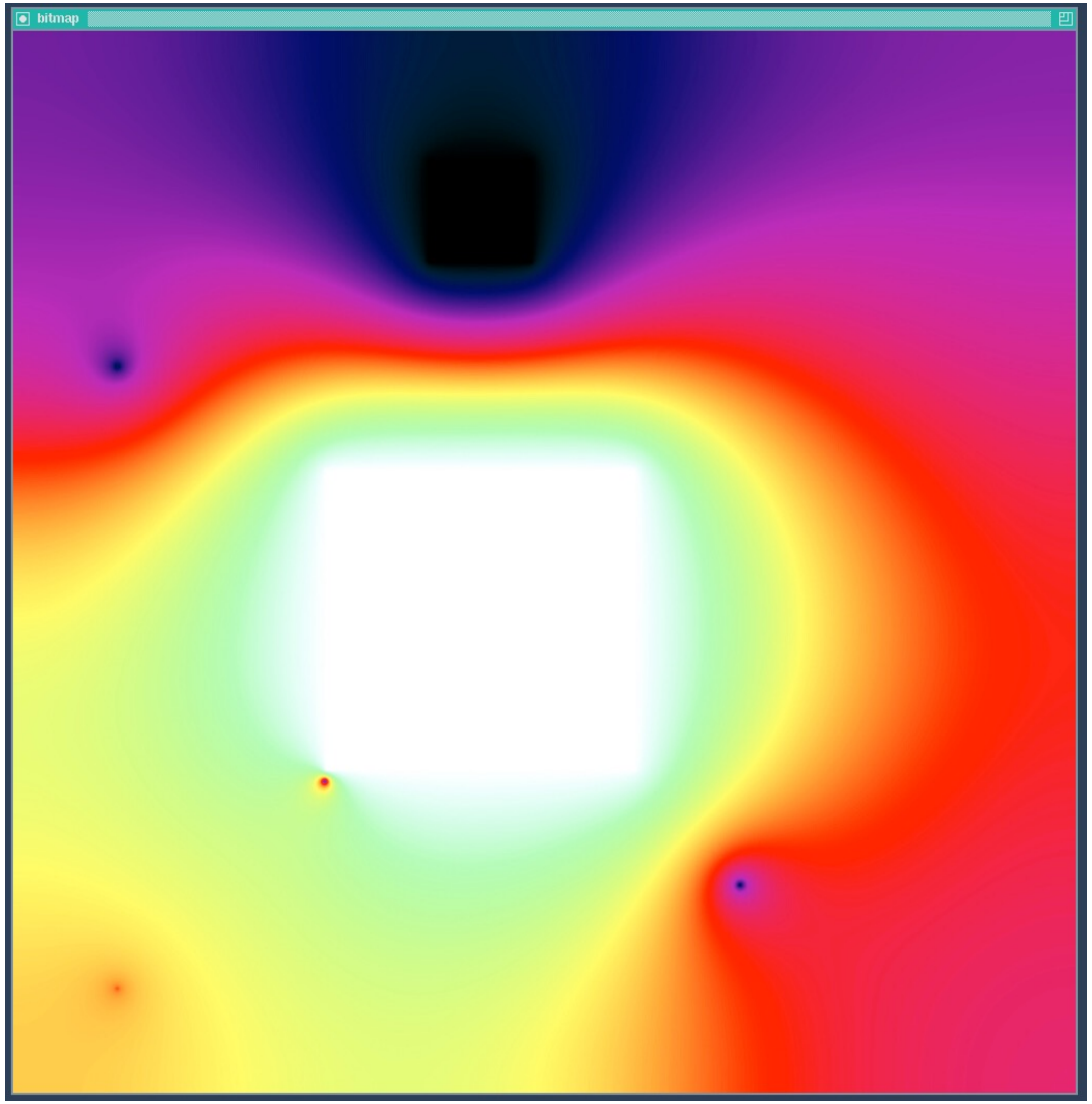
- Add a dimensionality argument of 2 to declare two-dimensional textures
  - texture<float, 2> texConstSrc;
  - texture<float, 2> texIn;
  - texture<float, 2> texOut;
- Change tex1Dfetch() calls to text2D() calls
- Provide a cudaChannelFormatDesc when binding two-dimensional textures

```
float   t, l, c, r, b;
if (dstOut) {
    t = tex2D(texIn,x,y-1);
    l = tex2D(texIn,x-1,y);
    c = tex2D(texIn,x,y);
    r = tex2D(texIn,x+1,y);
    b = tex2D(texIn,x,y+1);
} else {
    t = tex2D(texOut,x,y-1);
    l = tex2D(texOut,x-1,y);
    c = tex2D(texOut,x,y);
    r = tex2D(texOut,x+1,y);
    b = tex2D(texOut,x,y+1);
}
dst[offset] = c + SPEED * (t + b + r + l - 4 * c);
```

```
float c = tex2D(texConstSrc,x,y);
if (c != 0)
    iptr[offset] = c;
```

```
cudaChannelFormatDesc desc = cudaCreateChannelDesc<float>();
```

Heat Transfer -- Screenshot



1D Texture vs. 2D Texture?

- The version of our heat transfer simulation that uses two-dimensional textures has essentially identical performance characteristics as the version that uses one-dimensional textures
- From a performance standpoint, the decision between one- and two-dimensional textures is likely to be inconsequential
- Make the decision between one- and two-dimensional textures on a case-by-case basis

Takeaways

- Global memory is large but slow
- Shared memory is small but fast, great for applications that exhibit locality of data access
- Use constant memory for data that will not change over the course of a kernel execution to reduce the required memory bandwidth
- Use texture memory for application where memory access patterns exhibit a great deal of “spatial locality” – the use of data elements within relatively close storage locations