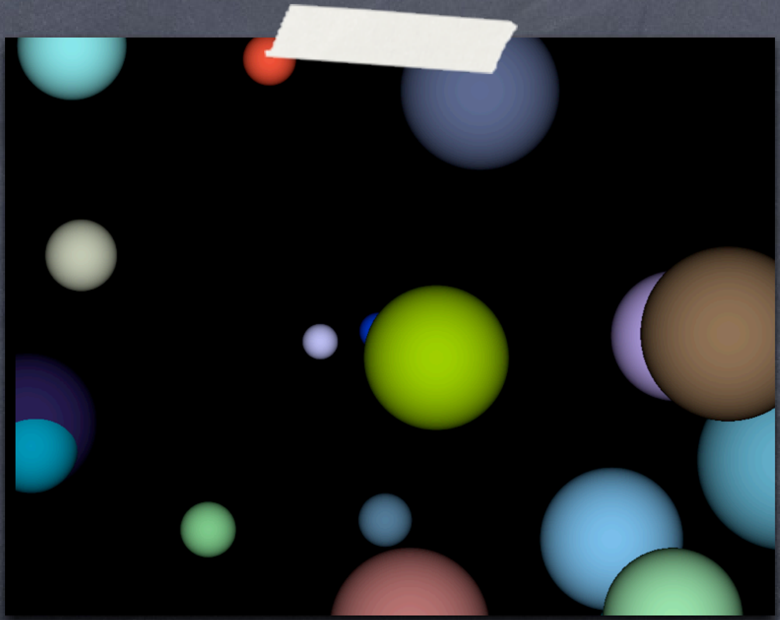
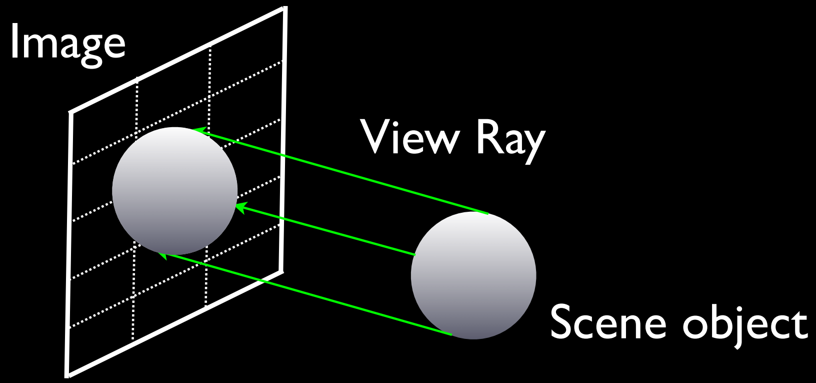
Constant Memory and Events

Screenshot from the ray tracing example



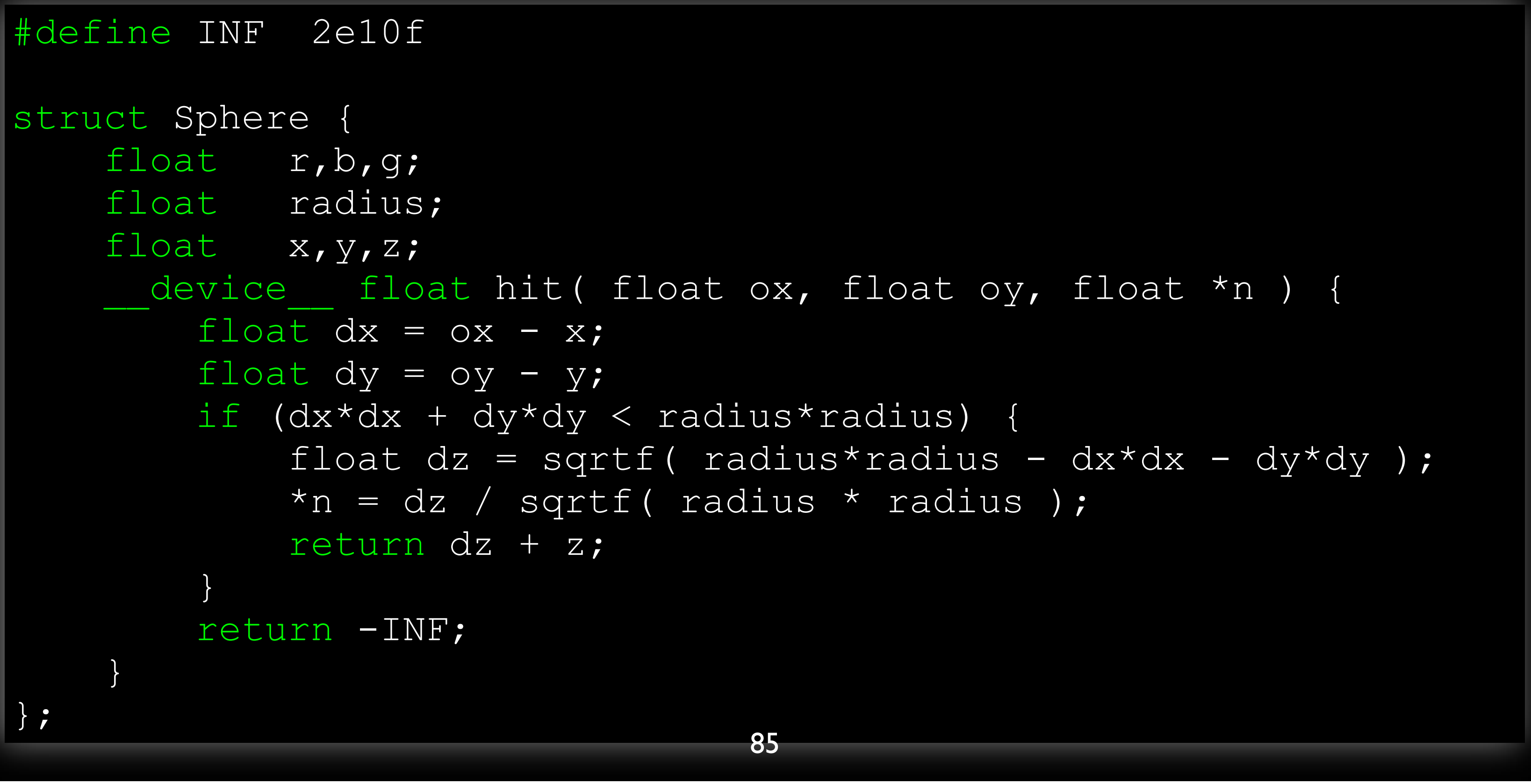
Problem: Memory

- GPUs: Enormous amount of arithmetic power
- Bottleneck: memory bandwidth
- **Constant memory** instead of *"shared memory"*
- Example: ray tracing



Basic ray tracer on GPU

- Extremely simple:
  - no lightning
  - only spheres
  - hides surfaces not seen by the camera



```
#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;
    }
};
```

Basic ray tracer (main)

```
#include "cuda.h"
#include "../common/book.h"
#include "../common/cpu_bitmap.h"
#define rnd(x) (x * rand() / RAND_MAX)
#define SPHERES 20

int main( void ) {
   DataBlock   data;
   // capture the start time
   cudaEvent_t     start, stop;
   HANDLE_ERROR( cudaEventCreate( &start ) );
   HANDLE_ERROR( cudaEventCreate( &stop ) );
   HANDLE_ERROR( cudaEventRecord( start, 0 ) );

   CPUBitmap bitmap( DIM, DIM, &data );
   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));
```

Basic ray tracer (main-2)

```
// 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 );
```

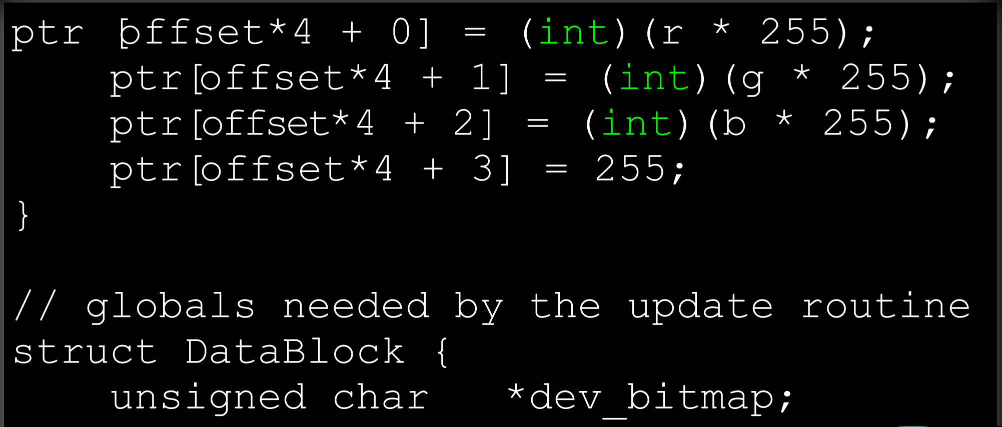
Basic ray tracer (main-3)

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

    // 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 ) );

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

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

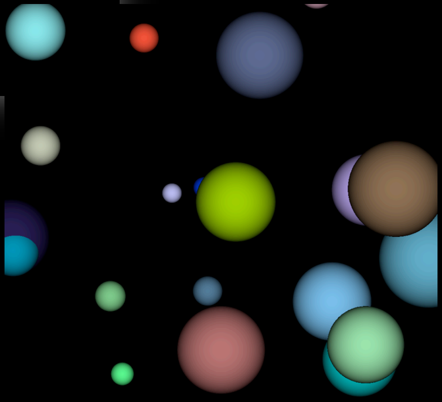
Basic ray tracer (kernel)

```
#define SPHERES 20

__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;
        }
    }
```

Basic ray tracer (6)



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

// globals needed by the update routine
struct DataBlock {
    unsigned char   *dev_bitmap;
    Sphere          *s;
};
```



Ray tracer with constant memory

Instead of declaring

```
Sphere          *s;
```

we add __constant__ before it:

```
__constant__ Sphere s[SPHERES];
```

Ray tracing with constant memory

```
#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;
    }
};
```

Ray tracing with constant memory (2)

```
int main( void ) {
    DataBlock   data;
    // capture the start time
    cudaEvent_t     start, stop;
    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );
    HANDLE_ERROR( cudaEventRecord( start, 0 ) );

    CPUBitmap bitmap( DIM, DIM, &data );
    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 ) );
```

Ray tracing with constant memory (3)

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

Ray tracing with constant memory (4)

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

    // 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 ) );

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

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

Ray tracing with constant memory (5)

```
#define SPHERES 20
__constant__ Sphere s[SPHERES];

__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;
        }
    }
```

Measuring the performance with Events

Creation and recording events:

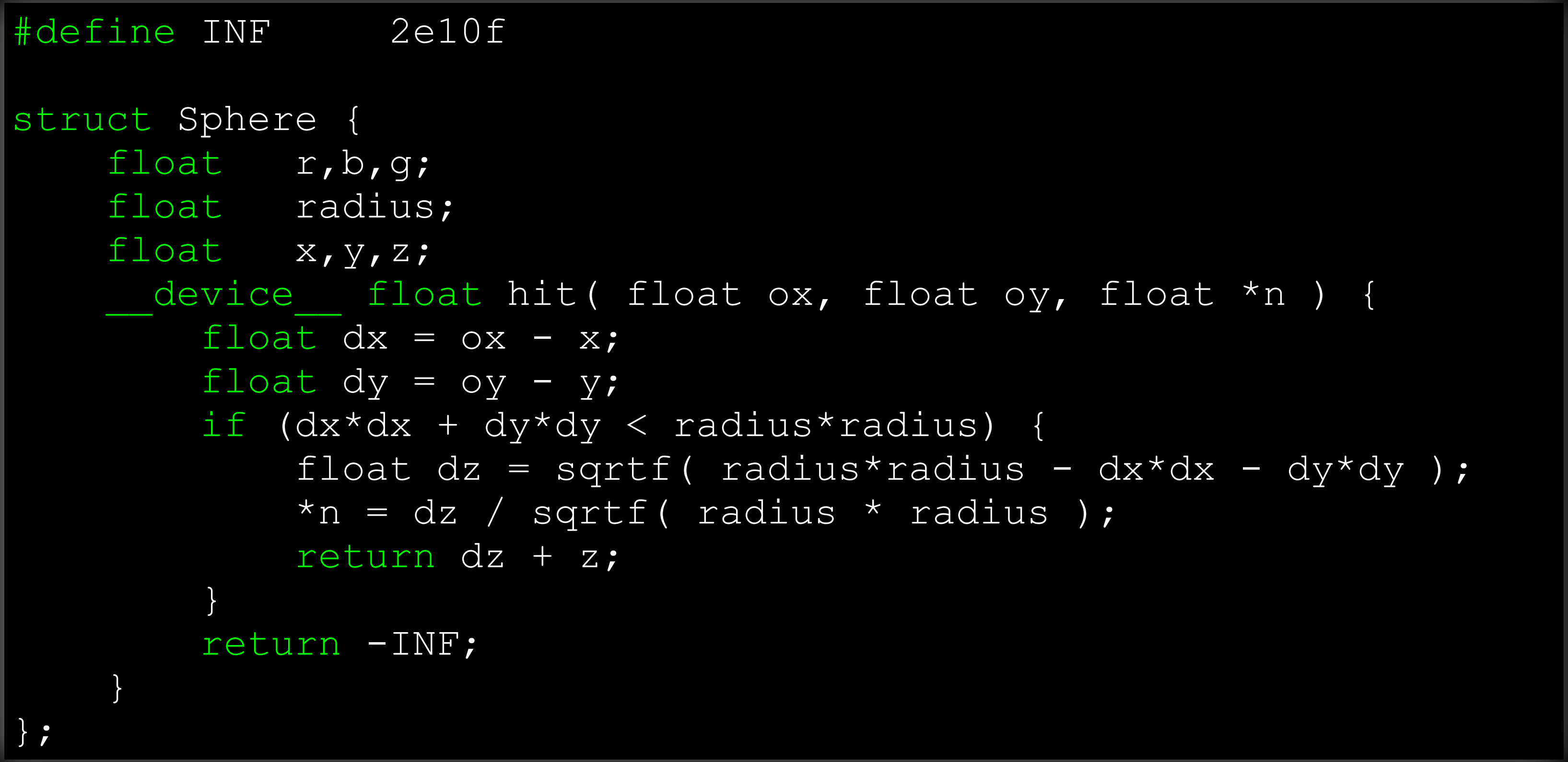
```
cudaEvent_t  start,stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start,0);

   // do some work on the GPU

cudaEventRecord(stop,0);
```

Measuring the performance with Events

Creation and recording events:

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

   // do some work on the GPU

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

Code without constant memory



```
#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;
    }
};
```

Code without constant memory

```
int main( void ) {
    DataBlock   data;
    // capture the start time
    cudaEvent_t     start, stop;
    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );
    HANDLE_ERROR( cudaEventRecord( start, 0 ) );

    CPUBitmap bitmap( DIM, DIM, &data );
    unsigned char   *dev_bitmap;

    // 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 ) );
```

Code without constant memory

```
// 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 );
```

Code without constant memory

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

    // 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 ) );

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

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

Code without constant memory

```
#define SPHERES 20

__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;
        }
    }
```

Code with constant memory

```
#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;
    }
};
```

Code with constant memory

```
int main( void ) {
    DataBlock   data;
    // capture the start time
    cudaEvent_t     start, stop;
    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );
    HANDLE_ERROR( cudaEventRecord( start, 0 ) );

    CPUBitmap bitmap( DIM, DIM, &data );
    unsigned char   *dev_bitmap;



    // allocate memory on the GPU for the output bitmap
    HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap,
                              bitmap.image_size() ) );
```

Code with constant memory

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

Code with constant memory

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

    // 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 ) );

    HANDLE_ERROR( cudaFree( dev_bitmap ) );

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

Code with/without constant memory

Exercice:

- Every team records the performances of the two different versions of the code and give me the results

Review

Suggestion: try the "CUDA by example" code, chap. 6

- NVIDIA HW makes other types of memory available for us

- We also get additional constraints to use this memory

- We have seen how to use CUDA events to record the performance of our code