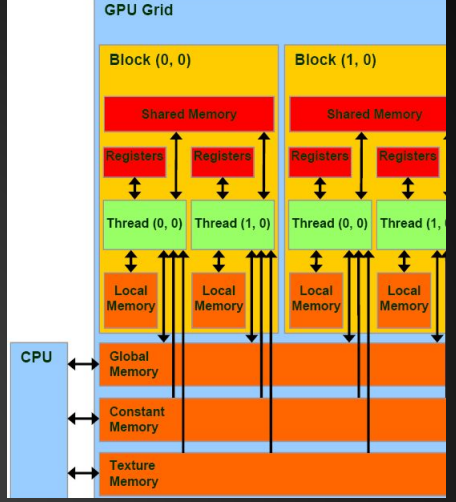
Universidad Católica San Pablo

CUDA **Constant Memory**

Marc-Antoine Le Guen

mleguen@ucsp.edu.pe

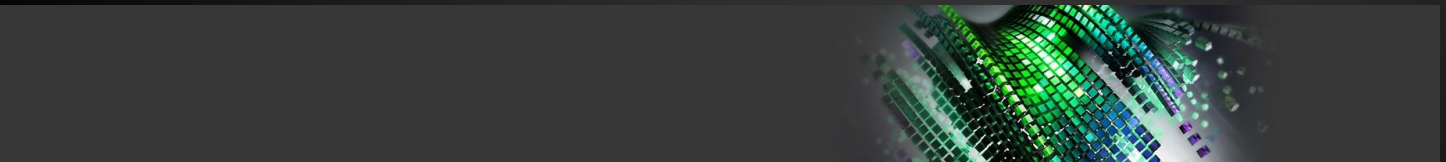
Constant memory



- `__constant__`
- Read-only
- 64 KB
- Reduce el uso del ancho de banda de la memoria
- Porque es más rápido ?
  - Una lectura de esta memoria por un thread puede ser transmitida a 15 thread más
    - half-warp of thread : 16 threads
    - 94 % de ganancia de tiempo de acceso
  - La memoria constante está almacenada en un cache para acelerar otros accesos
    - Se puede almacenar en caché porque estos datos son constantes
    - Acceso acelerado para los otros half-warps

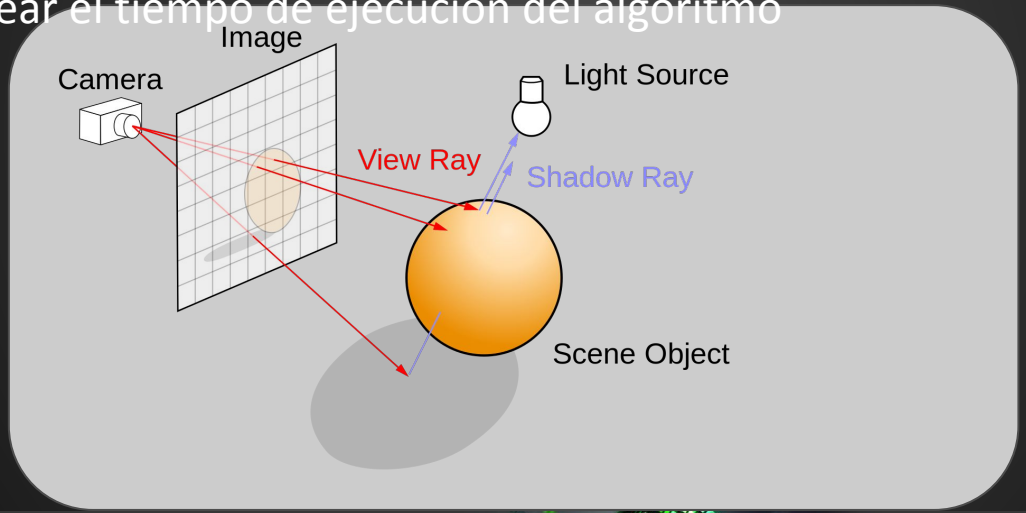
Constant memory

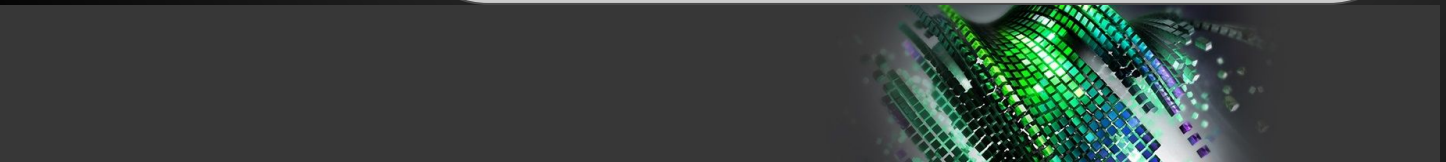
- Peligro de la memoria constante
  - Cada thread quiere acceder a un dato diferente de la memoria constante.
    - Tiempo del proceso > tiempo de acceso a la memoria global
- Es importante poder monitorear el tiempo de ejecución de una porción de código CUDA



Constant memory - Basic Ray Tracing

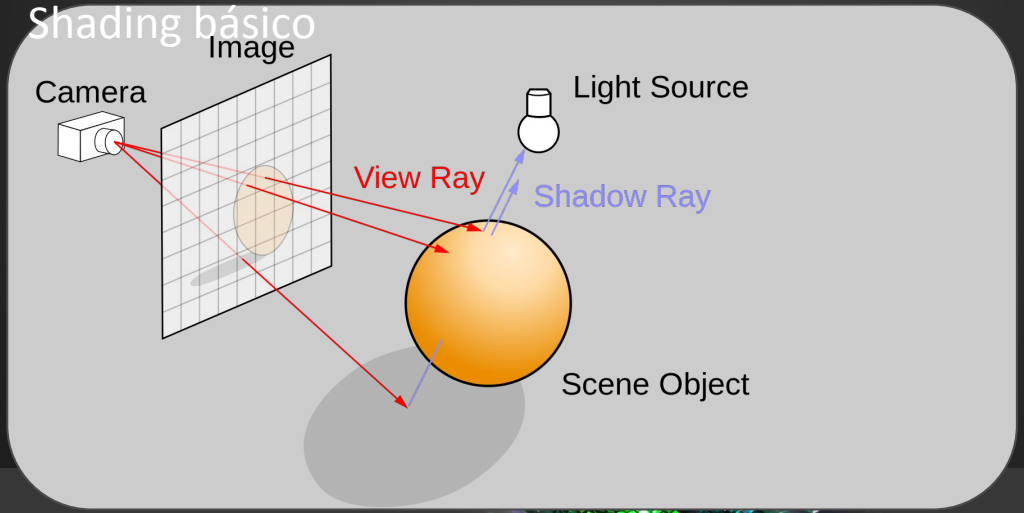
- Implementación de un sistema de Ray Tracing básico que usa la memoria constante
- Monitorear el tiempo de ejecución del algoritmo





Constant memory - Basic Ray Tracing

- Simplificamos
  - Lanzar un rayo por píxel definido por su posición (x,y)
  - No utilizaremos la luz
    - Shading básico



Constant memory - Basic Ray Tracing

- Simplificamos
  - Lanzar un rayo por pixel definido por su posición (x,y)
  - No utilizaremos la luz

Estructura de una esfera

```
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;
        }
};
```

- r,g,b color
- radio
- posición 3D x,y,z
- hit() función retorna a que profundidad tocó el rayo que partió de un pixel i,j

Constant memory - Basic Ray Tracing

Kernel

```
__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;
}
```

- Recuperamos x,y nuestra posición en la imagen
- calculamos offset para escribir en la imagen
- calculamos ox,oy la posición de pixel en el espacio
- Iteramos sobre la esferas para buscar si el rayo cruza alguna y calculamos el color que corresponde.
- Escribimos en la imagen el color que calculamos previamente (negro si nada)

Constant memory - Basic Ray Tracing

Sin constant memory

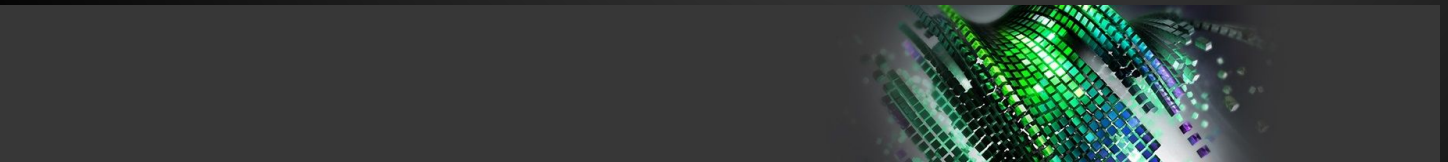
```
#define rnd( x ) (x * rand() / RAND_MAX)
#define INF     2e10f
#define SPHERES 20

// main
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;
        }
        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, gpu_bitmap );
```

Con constant memory

```
#define rnd( x ) (x * rand() / RAND_MAX)
#define INF     2e10f
#define SPHERES 20
__constant__ Sphere s[SPHERES];

//main()
// allocate temp memory, initialize it, copy to constant
// 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;
        }
        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>>>( gpu_bitmap );
```

Constant memory - Basic Ray Tracing

- Asignación estática
  - `cudaMalloc( (void**)&s, sizeof(Sphere) * SPHERES );`
  - `__constant__ Sphere s[SPHERES];`
- Copia a la memoria constante
  - `cudaMemcpy( s, temp_s,sizeof(Sphere) * SPHERES, cudaMemcpyHostToDevice );`
  - `cudaMemcpyToSymbol( s, temp_s, sizeof(Sphere) * SPHERES);`
- Kernell call
  - `kernel<<<grids,threads>>>( s, gpu_bitmap );`
  - `kernel<<<grids,threads>>>( gpu_bitmap );`



Constant memory - Basic Ray Tracing

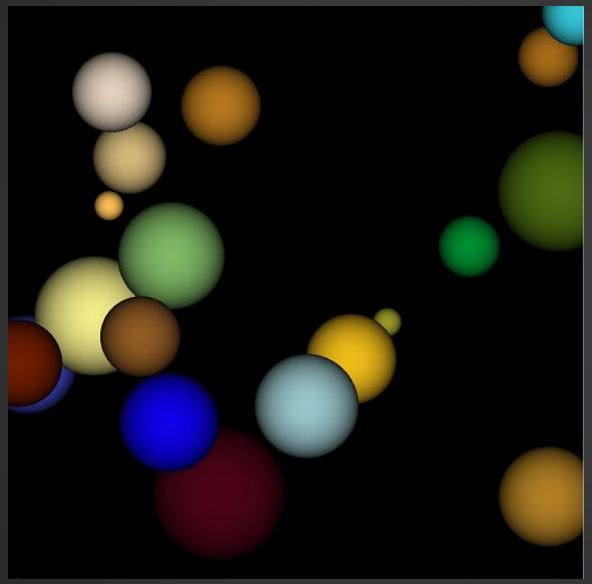
Kernel - Sin constant memory

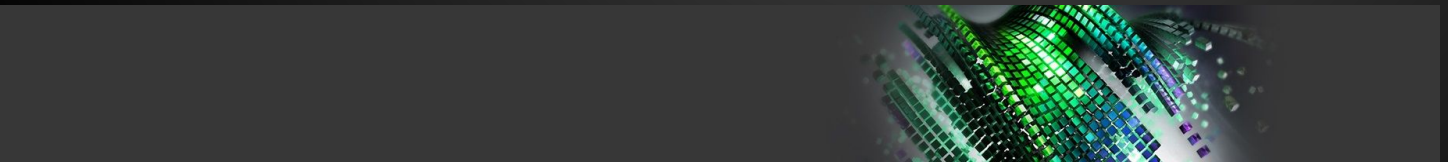
```
__global__ void kernel( Sphere *s, unsigned char *ptr ) {
        // map from threadIdx/BlockIdx to pixel position
        int x = threadIdx.x + blockIdx.x * blockDim.x;
        int y = threadIdx.y + blockIdx.y * blockDim.y;
        int offset = x + y * blockDim.x * gridDim.x;
        float   ox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

Kernel - Con constant memory

```
__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;
        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;
}
```

Constant memory - Basic Ray Tracing

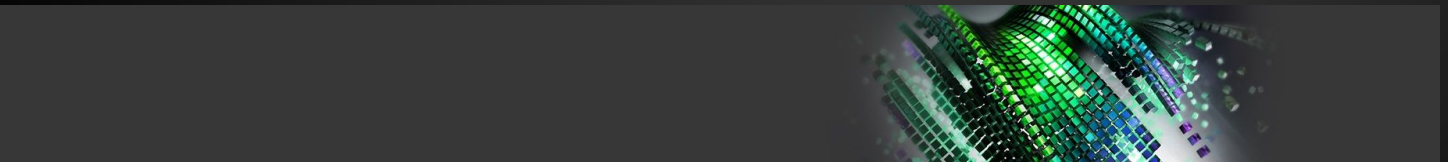




Monitorear código

- Obtener el tiempo de ejecución de una porción de código
  - Crear *events*
- Calcular el tiempo entre los dos eventos

```
cudaEvent_t start , stop;
cudaEventCreate (&start);
cudaEventCreate (&stop);
cudaEventRecord ( start, 0 );
// do some work on the GPU

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

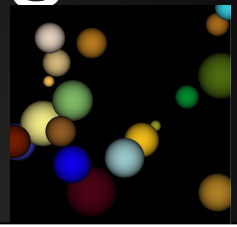


Monitorear código

- Obtener el tiempo de ejecución de una porción de código
  - Crear *events*
- Calcular el tiempo entre los dos eventos

```
cudaEvent_t start , stop;
cudaEventCreate (&start);
cudaEventCreate (&stop);
cudaEventRecord ( start, 0 );
// do some work on the GPU

cudaEventRecord ( stop, 0 );
cudaEventSynchronize ( stop );
float   elapsedTime ;
cudaEventElapsedTime ( &elapsedTime , start, stop );
printf( "Time to generate:  %3.1f ms\n" , elapsedTime );
```

Monitorear - Basic Ray Tracing



- Sin memoria constante : 3.8 ms 670 GTX
- Con memoria constante : 3.0 ms 670 GTX
- [illegible] 27 %

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

    unsigned char   *gpu_bitmap;

    // allocate memory on the GPU for the output bitmap
    cudaMalloc( (void**)&gpu_bitmap,DIM*DIM*4);

    // allocate temp memory, initialize it, copy to constant
    // 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;
    }
```

```
        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>>>( gpu_bitmap );

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

    cudaEventDestroy( start );
    cudaEventDestroy( stop );

    cudaFree( gpu_bitmap );
```