Texture Memory

Texture Memory

- read-only memory
- Can improve performance and reduce memory traffic when reads have certain access patterns.
- Originally designed for the classical OpenGL and DirectX rendering pipelines.
- But has some properties that make it extremely useful for computing, especially computer vision application.

Texture Memory

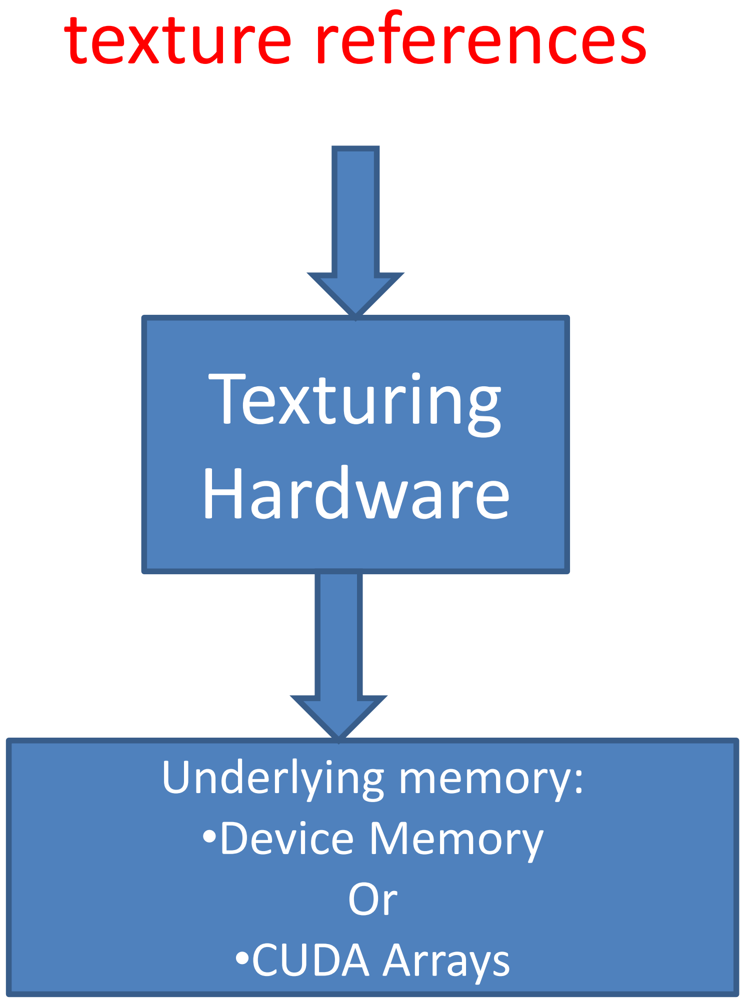
- Texture memory is cached on chip
  - In KB range in every SM
  - In some situations it will provide higher effective bandwidth by reducing memory requests to off-chip DRAM.
- Texture caches are designed for graphics applications where memory access patterns exhibit a great deal of spatial locality.
- Every SM has several texture fetch units

One SM from Pascal architecture (GP100)

Source: https://www.extremetech.com/extreme/226032-nvidias-pascal-gp100-gpu-massive-bandwidth-enormous-double-precision-performance

Texture Memory

- The texture cache is optimized for 2D spatial locality.
- Part of DRAM
- The process of reading a texture is called a *texture fetch*.
- Can be addressed as 1D, 2D, or 3D dimensional arrays.
- Elements of the array are called *texels*.


texture references
Texturing Hardware
Underlying memory:
•Device Memory
Or
•CUDA Arrays

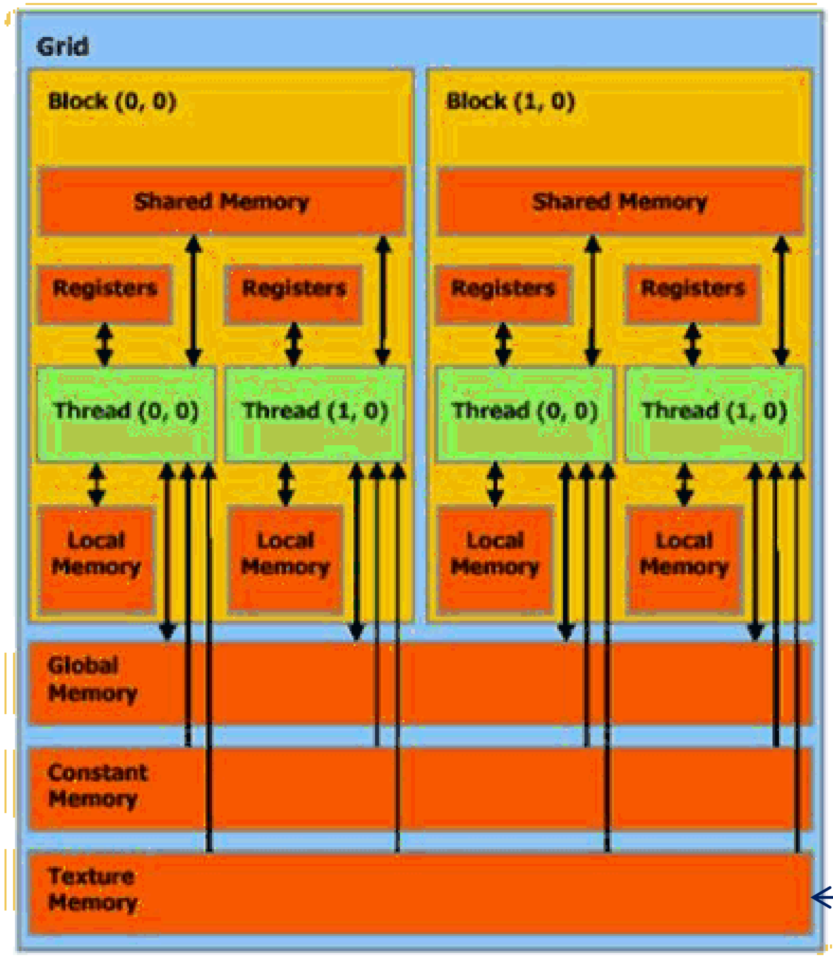
CUDA Arrays

- Designed specifically to support texturing.
- Allocated from device memory
- Do not consume any CUDA address space.
- Have an opaque layout
- Cannot be addressed by pointers
- Memory locations addressed using two things:
  - array handle
  - set of 1D, 2D, or 3D coordinates

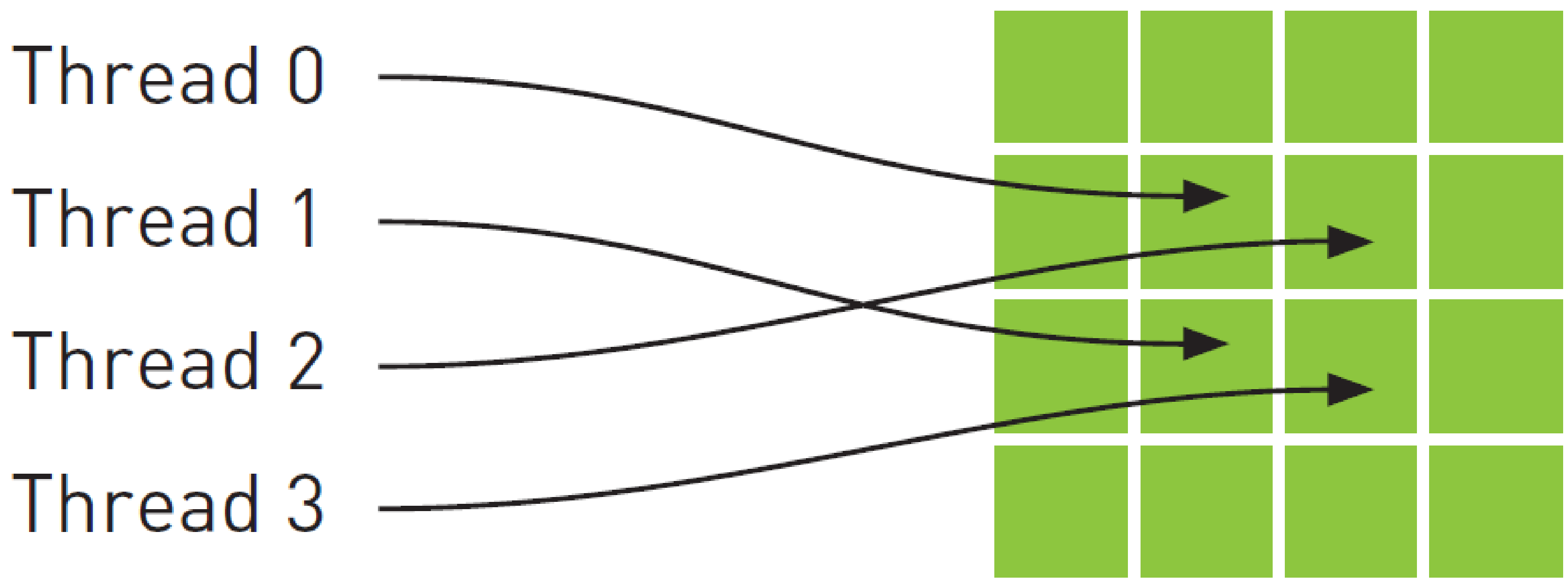
Why CUDA Arrays?

- Designed so that contiguous addresses exhibit 2D or 3D locality.

Texture Memory

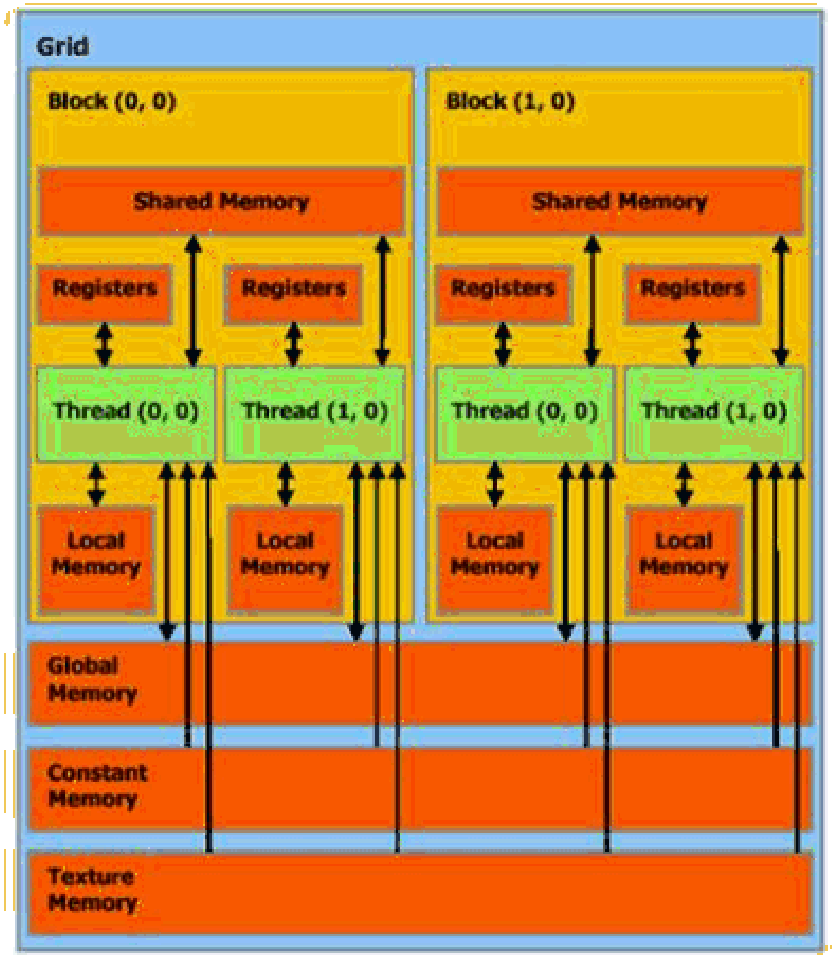


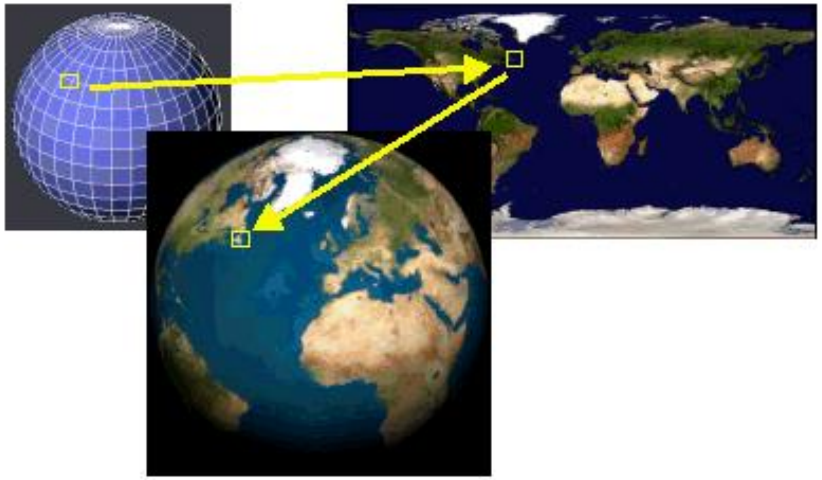
To accelerate frequently performed operations such as mapping a 2D "skin" onto a 3D polygonal model.



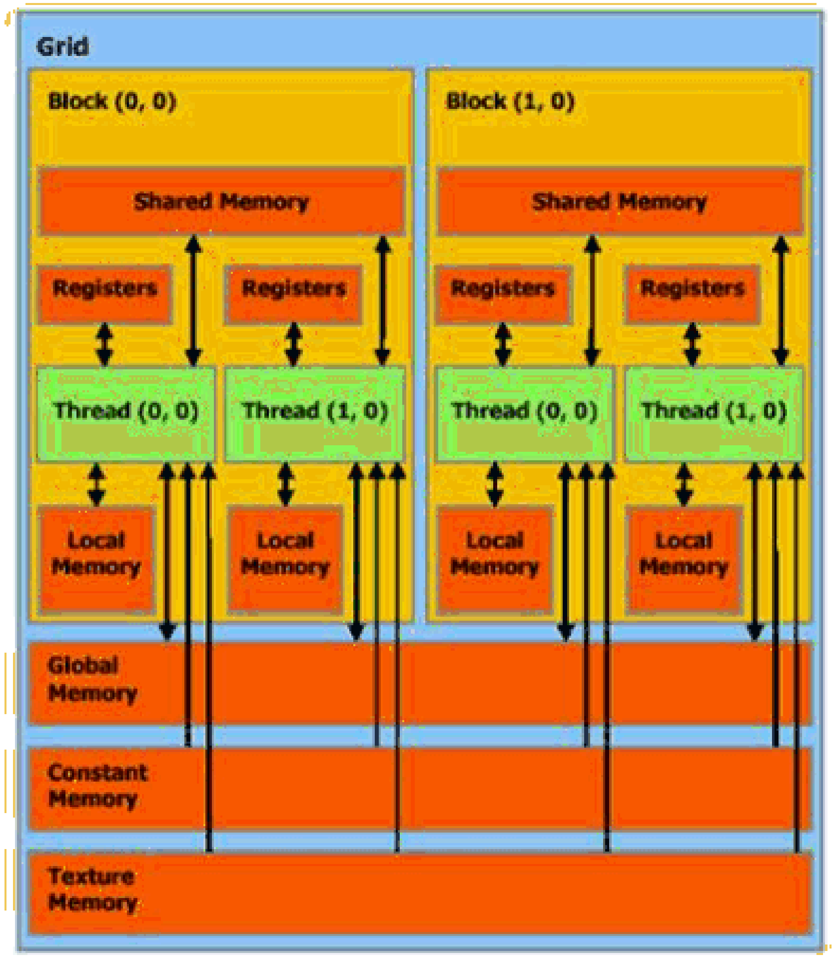
source: http://cuda-programming.blogspot.com/2013/02/texture-memory-in-cuda-what-is-texture.html

Texture Memory





Texture Memory



Capabilities:

- Ability to cache global memory
- Dedicated interpolation hardware
- Provides a way to interact with the display capabilities of the GPU.

The best performance is achieved when the threads of a warp read locations that are close together from a spatial locality perspective.

Allocating CUDA Arrays

```
cudaError_t cudaMallocArray (
        struct cudaArray **          array,
        const struct cudaChannelFormatDesc *     desc,
        size_t     width,
        size_t     height,
        unsigned int  flags
);
```

array: pointer to allocated array in device memory
width: array width in bytes
height: default is 0 → 1D array
flags:

- cudaArrayDefault: default array allocation
- cudaArraySurfaceLoadStore

Allocating CUDA Arrays

```
cudaError_t cudaMallocArray          (
            struct cudaArray **  array,
            const struct cudaChannelFormatDesc *  desc,
            size_t      width,
            size_t      height,
            unsigned int  flags
);
```

```
struct cudaChannelFormatDesc
cudaCreateChannelDesc(x,y,z,w,f);
```

```
struct cudaChannelFormatDesc {
    int x, y, z, w;   ← number of bits in each member of the texture element
    enum cudaChannelFormatKind f;
  };
```

cudaChannelFormatKindSigned	Signed channel format
cudaChannelFormatKindUnsigned	Unsigned channel format
cudaChannelFormatKindFloat	Float channel format
cudaChannelFormatKindNone	No channel format

Texture Fetch

- First parameter is texture reference
  - defines which part of texture memory is fetched
  - must be bound through runtime functions to texture memory
  - Attribute:
    - texture is addressed as 1D, 2D, or 3D
    - the input and output data types of the texture fetch
    - the input coordinates are interpreted
    - what processing should be done
  - Type of texels are the basic: integer, single/double precision floating point, … .

Steps for Using Texture Memory in Your CUDA Code

1. Declare the texture memory in CUDA.
2. Bind the texture memory to your texture reference in CUDA.
3. Read the texture memory from your texture reference in CUDA Kernel.
4. Unbind the texture memory from your texture reference in CUDA.

Step 1: Declare

texture <type, dim, readmode> texture_reference;

- texture_reference: the handle to be used
- type: type of texel data returned from an access to the texture: int, float, … .
- dim: 1 (default), 2, or 3
- readmode: controls conversion of texel returned by an access
  - cudaReadModeElementType (default) no conversion
  - cudeReadModeNormalizedFloat
    - if type is integer, value returned is mapped to [-1.0,1.0] for signed, and [0.0, 1.0] for unsigned
- Example:

texture <float, 2, cudaReadModeElementType> mytex;

Step 2: Bind

cudaBindtexture (size *t offset,
& testure_reference , const void * devptr,
size_t size) ;

- Binds size bytes of the memory area pointed to by devPtr to texture reference texture_reference.
- offset parameter is an optional byte offset.
- devPtr: Memory area on device
- size: Size of the memory area pointed to by devPtr

Step 3: Read

- The easiest is: tex1Dfetch()

Example:

```
texture <int,1,cudaReadModeElementType> texref;
__global__
void textureTest(int *out){
    int  tid =  blockIdx.x * blockDim.x + threadIdx.x;
    float x;
    int i;
    for(i=0; i<30; i++)
        x = tex1Dfetch(texref, i);
}
```

Step 4: Unbind

```
cudaUnbindTexture(texture_reference);
```

So

- Texture memory size is very small.
- We just scratched the surface of texture memory.
- Two usages of texture memory outside graphics applications:
  - Using texture cache to reduce bandwidth and work around coalescing constraints.
  - Make use of advanced fixed-function hardware put inside GPU for graphics applications