Demand-Loaded Textures in OptiX 7

June 2019

Motivation

Support 1 TB+ of textures in OptiX

- Film renders often load 10-50 GB of texture tiles.
    - Total texture file size is often 200 - 500 GB.

- Loading entire textures wastes memory, bandwidth, and I/O.
    - Should allocate / load only the miplevels that are required.
    - Or individual tiles, better yet.

Miplevels vs. tiles

Allocation vs. filling

- Sparse textures (tiled resources in DX) are supported by hardware.
    - But not yet exposed in CUDA.
    - Performance and flexibility are uncertain.

- Interim approach: allocate and fill entire miplevels
    - Truncated pyramid: one contiguous allocation spanning required miplevels
    - Adjacent miplevels are contiguous in memory
    - Advantage: fully hardware accelerated (including anisotropic filtering)
    - Reallocate backing storage as new levels are requested.

Approach

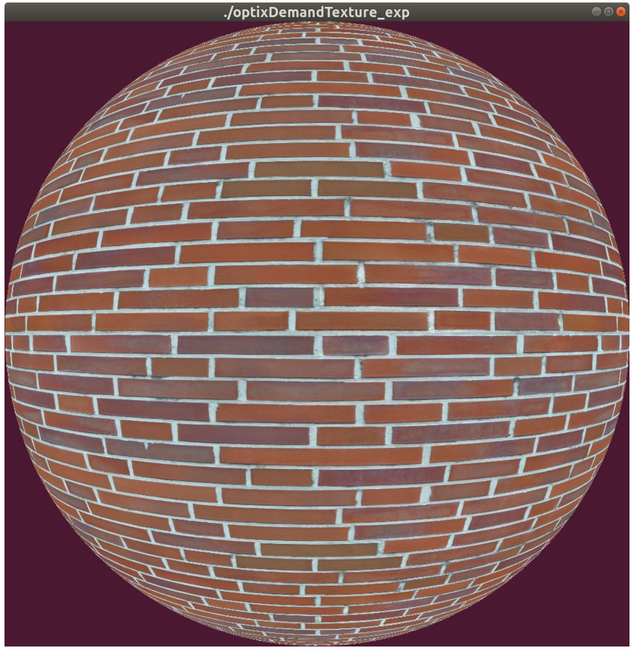
Cooperative paging

- If a required miplevel is not resident when fetching from demand-loaded texture:
  - Page request is recorded.
  - Shader can throw a user exception, terminating the ray (but not the launch).
  - Alternatively, can substitute a default color and continue (for progressive refinement).
- Upon launch completion, fill requested miplevels from disk or host memory.
  - Page table is updated on device.
- Kernel is relaunched and failed rays are recast.

Demand-loaded texture sample for OptiX 7

optixDemandTexture

- Adapted from optixSphere sample.
- Encapsulates low-level OptiX paging routines.
  - All user-level code. Not part of OptiX 7 API.
- Main classes
  - DemandTextureManager
  - DemandTexture
  - EXRImage
- Focus on simplicity
  - Synchronous
  - Single GPU
  - No eviction
- Good starting point for production-ready GPU texture system.



Summary

- Construct DemandTextureManager
  - Initialize low-level OptiX paging library.
- Create DemandTexture
  - Construct image loader.
  - Construct sampler (cudaTextureObject).
- Prepare for launch
  - Copy sampler array to device.
  - Pass sampler array as launch parameter.
  - Pass texture id as HitGroupData.
- In closest hit shader:
  - Use texture id to obtain sampler.
  - Use gradients to calculate miplevel
  - Request required miplevels.
  - If resident, fetch from texture.
- Process requests
  - Scan usage/residence bits for requested pages:

    `requested = used && !resident`
  - Determine min and max miplevels per texture.
  - Reallocate textures and update samplers.
  - Fill requested miplevels.
  - Update residence bits.
- Relaunch after processing requests
  - Copy sampler array to device.

DemandTextureManager

Encapsulates OptixPaging library

```
class DemandTextureManager
{
  private:
    // The OptiX paging system employs a context that includes the page table, etc.
    OptixPagingContext* m_pagingContext = nullptr;

    // Device memory used to call OptiX paging library routines.
    // These allocations are retained to reduce allocation overhead.
    uint32_t* m_devRequestedPages    = nullptr;
    uint32_t* m_devNumPagesReturned  = nullptr;
    MapType*  m_devFilledPages       = nullptr;
};
```

Construct DemandTextureManager

Initialize the OptiX paging library.

```
DemandTextureManager::DemandTextureManager()
{
    // Configure the paging library.
    OptixPagingOptions options{NUM_PAGES, NUM_PAGES};
    optixPagingCreate( &options, &m_pagingContext );

    OptixPagingSizes sizes{};
    optixPagingCalculateSizes( options.initialVaSizeInPages, sizes );

    // Allocate device memory required by the paging library.
    CUDA_CHECK( cudaMalloc( reinterpret_cast<void**>( &m_pagingContext->pageTable ),
                            sizes.pageTableSizeInBytes ) );
    CUDA_CHECK( cudaMalloc( reinterpret_cast<void**>( &m_pagingContext->usageBits ),
                            sizes.usageBitsSizeInBytes ) );
    optixPagingSetup( m_pagingContext, sizes, 1 );

    // Allocate device memory that is used to call paging library routines.
    // These allocations are retained to reduce allocation overhead.
    CUDA_CHECK( cudaMalloc( &m_devRequestedPages, MAX_REQUESTED_PAGES * sizeof( uint32_t ) ) );
    CUDA_CHECK( cudaMalloc( &m_devNumPagesReturned, 3 * sizeof( uint32_t ) ) );
    CUDA_CHECK( cudaMalloc( &m_devFilledPages, MAX_NUM_FILLED_PAGES * sizeof( MapType ) ) );
}
```

Image loader fills miplevels on demand

MipmappedImage (abstract base class)

```
class MipmappedImage
{
  public:
    /// Image info, including dimensions and format.
    struct Info
    {
        unsigned int width;
        unsigned int height;
        unsigned int elementSize;
        unsigned int numMipLevels;
    };

    /// The destructor is virtual to ensure that instances of derived classes are properly destroyed.
    virtual ~MipmappedImage() {}

    /// Open image and read header, returning info via the result parameter.  Returns false on error.
    virtual bool open( Info* info ) = 0;

    /// Read the specified miplevel into the given CUDA array.
    virtual bool read( cudaArray_t dest, unsigned int miplevel,
                       unsigned int width, unsigned int height ) = 0;
};
```

OpenEXR image loader

EXRImage implements MipmappedImage

- Constructor copies filename.
- open() method reads dimensions etc. from EXR header.
  - Lazily initialized when texture is first used.
- read() method reads a miplevel into device memory.
  - Uses OpenEXR library (Imf::TiledInputFile)
  - Calls cudaMemcpy2DToArray.
    - Currently synchronous. Async sample / library is anticipated
- (The read method is analogous to the callback on a demand-loaded buffer in OptiX 6)

Create a demand-loaded texture

DemandTextureManager::createTexture

- DemandTexture constructed from MipmappedImage.
- Assigned texture identifier, which is index into DemandTexture array (on host).
- Create pageId from texture id and miplevel number.
- Create texture sampler, which contains cudaTextureObject.
    - DemandTextureSampler array will be maintained on the device (indexed by texture id).

Create a demand-loaded texture

DemandTextureManager::createTexture

```
// Create a demand-loaded texture with the specified dimensions and format.  The texture initially has no
// backing storage.
const DemandTexture& DemandTextureManager:: createTexture ( std::shared_ptr<MipmappedImage>  image )
{
    // Add new texture to the end of the list of textures.  The texture identifier is simply its
    // index in the DemandTexture array, which also serves as an index into the device-side
    // DemandTextureSampler array.  The texture holds a pointer to the image, from which miplevel
    // data is obtained on demand.
    unsigned int textureId = static_cast<unsigned int>( m_textures.size() );
    m_textures.emplace_back( DemandTexture( textureId, image ) );
    DemandTexture& texture =  m_textures.back();

    // Create texture sampler, which will be synched to the device in launchPrepare().  Note that we
    // don't set m_hostSamplersDirty when adding new samplers.
    m_hostSamplers.emplace_back( texture.getSampler() );

    return texture;
}


struct DemandTextureSampler
{
    cudaTextureObject_t  texture;
};
```

Prepare for launch

Copy DemandTextureSampler array to device.

```
void DemandTextureManager::launchPrepare()
{
    ...
    // Reallocate device sampler array.
    DemandTextureSampler* oldSamplers = m_devSamplers;
    CUDA_CHECK( cudaMalloc( reinterpret_cast<void**>( &m_devSamplers ),
                            m_textures.size() * sizeof( DemandTextureSampler ) ) );

    // If any samplers are dirty (e.g. textures were reallocated), copy them all from the host.
    if( m_hostSamplersDirty )
    {
        CUDA_CHECK( cudaMemcpy( m_devSamplers, m_hostSamplers.data(),
                                m_hostSamplers.size() * sizeof( DemandTextureSampler ),
                               cudaMemcpyHostToDevice ) );
        m_hostSamplersDirty = false;
    }
    else
    {
        // Otherwise copy the old samplers from device memory and the new samplers from host memory.
        ...
    }
    ...
}
```

Launch parameters

OptixPagingContext and DemandTextureSampler array

```
struct Params
{
    uchar4*                        image;
    uint32_t                       image_width;
    uint32_t                       image_height;
    // ...
    OptixPagingContext             pagingContext;
    const DemandTextureSampler*    demandTextures;
};
```

Hit group data

Provides texture id to closest hit shader

```
struct HitGroupData
{
    float    radius;
    uint32_t demand_texture_id;
};

...

HitGroupSbtRecord hg_sbt;
hg_sbt.data = { 1.5f /*radius*/, texture.getId() };

OPTIX_CHECK( optixSbtRecordPackHeader( hitgroup_prog_group, &hg_sbt ) );
CUDA_CHECK( cudaMemcpy( reinterpret_cast<void*>( hitgroup_record ),
                        &hg_sbt, hitgroup_record_size, cudaMemcpyHostToDevice ) );
```

Closest hit shader

Index sampler array with texture id

```
HitGroupData* hg_data =
    reinterpret_cast<HitGroupData*>( optixGetSbtDataPointer() );

const DemandTextureSampler& sampler =
    params.demandTextures[hg_data->demand_texture_id];
```

Closest hit shader

Use gradients to calculate miplevel

```
// No sample code for this yet, but it goes like this (ignoring anisotropy)
__device__ float tex2DGradMipLevel ( int numMipLevels, float2 ddx, float2 ddy )
{
    // Filter width is minimum length of ddx, ddy.
    float lx          = sqrtf( ddx.x*ddx.x + ddx.y*ddx.y );
    float ly          = sqrtf( ddy.x*ddy.x + ddy.y*ddy.y );
    float filterWidth = fmin( lx, ly );

    // Choose the miplevel where the filter width covers two texels.
    return numMipLevels + log2( filterWidth ) - 1.0f;
}
```

Closest hit shader

Check whether required miplevels are resident

```
// Check whether a specified miplevel of a demand-loaded texture is resident,
// recording a request if not.
__device__ void requestMipLevel( unsigned int textureId, const DemandTextureSampler& sampler,
                                 unsigned int mipLevel, bool& isResident )
{
    // A page id consists of the texture id (upper 28 bits) and the miplevel number (lower 4 bits).
    const unsigned int requestedPage = textureId << 4 | mipLevel;

    // The paging context was provided as a launch parameter.
    const OptixPagingContext& context = params.pagingContext;

    // Check whether the requested page is resident, recording a request if not.
    optixPagingMapOrRequest( context.usageBits, context.residenceBits,
                             context.pageTable, requestedPage, &isResident );
}
```

Low-level OptiX paging library

optixPagingMapOrRequest

- Maintains device-side page table, residence bits, and usage bits (indexed by page id).
- Usage bit is set when page is referenced. Indicates page request if not resident.
- Residence bit determines whether page table entry is valid.
  - This permits concurrency without fine-grained synchronization.

```
__device__ uint64_t optixPagingMapOrRequest ( uint32_t* usageBits, uint32_t* residenceBits,
                                               uint64_t* pageTable, uint32_t pageId, bool* isValid )
{
    bool requested = checkBitSet( pageId, usageBits );
    if( !requested )
        atomicSetBit( pageId, usageBits );

    bool mapped = checkBitSet( pageId, residenceBits );
    *valid      = mapped;

    return mapped ? pageTable[pageId] :  0;
}
```

Get page requests from device

optixPagingPullRequests

- Kernel scans usage bit vector and resident bit vector (indexed by page id).
- A page is requested if it was used but not resident.
- Page requests are an array of page ids.

```
// Get page requests from the device (via optixPagingPullRequests).
void DemandTextureManager::pullRequests( std::vector<uint32_t>& requestedPages )
{
    // Get a list of requested page ids.  (Stale and evictable pages are currently unused).
    optixPagingPullRequests( m_pagingContext, m_devRequestedPages, MAX_REQUESTED_PAGES,
                             nullptr /*stalePages*/, 0, nullptr /*evictablePages*/,
                             0, m_devNumPagesReturned );
    ...
    // Copy the requested page list from this device.
    requestedPages.resize( numRequests );
    CUDA_CHECK( cudaMemcpy( requestedPages.data(), m_devRequestedPages,
                            numRequests * sizeof( uint32_t ), cudaMemcpyDeviceToHost ) );
}
```

Process page requests

DemandTextureManager::processRequests

- Sort requests by page number, grouping the requests for each texture.
- Determine min and max miplevel in each group of requests.
- Reallocate textures to accommodate new miplevels.
    - Lazily open image if necessary, reading dimensions etc. from header.
        - Virtual method call to MipmappedImage::open()
    - Copy existing miplevels (device to device).
    - Create new cudaTextureObject
- Update texture sampler (in host-side sampler array).
- Fill requested miplevels.
    - Virtual method call to MipmappedImage::read().
- Update residence bit vector on device (via optixPagingPushMappings)

Texture reallocation

Copy existing miplevels (device to device)

```
// Allocate new array.
cudaMipmappedArray_t  newMipLevelData;
CUDA_CHECK( cudaMallocMipmappedArray ( &newMipLevelData, &channelDesc, extent, numLevels ) );

...

// Copy any existing levels from the old array.
cudaMipmappedArray_t oldMipLevelData = m_mipLevelData;
m_mipLevelData = newMipLevelData;
for( unsigned int nominalLevel = oldMinMipLevel; nominalLevel <= oldMaxMipLevel; ++nominalLevel )
{
    unsigned int sourceLevel = nominalLevel - oldMinMipLevel;
    unsigned int destLevel   = nominalLevel - newMinMipLevel;
    ...
    CUDA_CHECK( cudaGetMipmappedArrayLevel ( &sourceArray, oldMipLevelData, sourceLevel ) );
    CUDA_CHECK( cudaGetMipmappedArrayLevel ( &destArray, newMipLevelData, destLevel ) );
    CUDA_CHECK( cudaMemcpy2DArrayToArray ( destArray, 0, 0, sourceArray, 0, 0, ... ) );
}
```

Texture reallocation

Create new cudaTextureObject

```
// Create resource description
cudaResourceDesc resDesc  = {};
resDesc.resType           = cudaResourceTypeMipmappedArray;
resDesc.res.mipmap.mipmap =  m_mipLevelData;
...

// Bias miplevel access in demand loaded texture based on the current minimum miplevel loaded.
texDesc.mipmapLevelBias = - static_cast<float>( m_minMipLevel  );

// Create texture object
CUDA_CHECK( cudaCreateTextureObject ( &m_texture, &resDesc, &texDesc,  nullptr ) );
```

Relaunch after processing requests

Sync sampler array to device

```
// Sync demand-texture sampler array to the device and provide it as a launch parameter.
textureManager.launchPrepare();
params.demandTextures = textureManager.getSamplers();

// The initial launch might accumulate texture requests.
optixLaunch( pipeline, stream, d_param, sizeof( Params ), &sbt, width, height, /*depth=*/1 );

// Repeatedly process any texture requests and relaunch until done.
for( int numFilled = textureManager.processRequests();
     numFilled > 0;
     numFilled = textureManager.processRequests() )
{
    // Sync sampler array and update launch parameter.
    textureManager.launchPrepare();
    params.demandTextures = textureManager.getSamplers();

    // Relaunch
    optixLaunch( pipeline, stream, d_param, sizeof( Params ), &sbt, width, height, /*depth=*/1 );
}
```

Summary

- Construct DemandTextureManager
  - Initialize low-level OptiX paging library.
- Create DemandTexture
  - Construct image loader.
  - Construct sampler (cudaTextureObject)
- Prepare for launch
  - Copy sampler array to device.
  - Pass sampler array as launch parameter.
  - Pass texture id as HitGroupData.
- In closest hit shader:
  - Use texture id to obtain sampler.
  - Use gradients to calculate miplevel
  - Request required miplevels.
  - If resident, fetch from texture.
- Process requests
  - Scan usage/residence bits for requested pages:

    ```
    requested = used && !resident
    ```
  - Determine min and max miplevels per texture.
  - Reallocate textures and update samplers.
  - Fill requested miplevels.
  - Update residence bits.
- Relaunch after processing requests
  - Copy sampler array to device.

Next steps

- Bug prevents use of tex2DLod and tex2DGrad (fix in progress).
- Sample needs ray differentials to drive miplevel selection.
- Fill only necessary tiles, rather than entire miplevel (in progress).
- Support multiple streams and multiple GPUs.
- Use asynchronous transfers.
- Test performance and scalability.

Future work

- Progressive refinement
  - Prefetch coarse miplevel(s).
  - Fall back to a different miplevel if available.
- Support eviction of least recently used miplevels.
  - Bucketed rendering might reduce texture working set size.
- Investigate software-based tiling for sparse textures.
  - Non-contiguous tiles, each with a border of texels from adjacent tiles for filtering.
- Anticipate CUDA sparse textures.