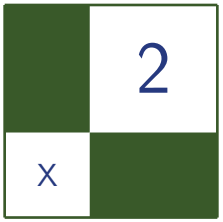

Accelerating Virtual Texturing Using CUDA

Charles-Frederik Hollemeersch, Bart Pieters, Peter Lambert, and Rik Van de Walle

For a long time games have used textures to add surface details and diversity to their virtual worlds. These textures usually consist of a basic set of different surface appearances which are then composed at runtime by shaders to calculate the final surface appearance. Recently there has been an increased interest in *virtual texturing* technologies [Barrett 08, Lefebre et al. 04, Mittring 08]. Virtual texturing allows very large textures (in the order of one gigapixel) to be applied to the game's geometry while still remaining within the limits of today's hardware. This allows far more varied worlds than can be achieved with composing tiling textures at a lower or comparable render cost.

One of the first commercial systems to employ these techniques was id Software's Mega Texturing technology [van Waveren 08] as used in the game *Enemy Territory Quake Wars*. This technology is loosely based on clipmapping [Tanner et al. 98] and shares the same limitation that only a single rectangle in texture space is available at the highest resolution. Hence its use is limited to nearly planar geometries with regular texture coordinates. In their upcoming game *Rage*, these limitations have been addressed by adopting a fully functional virtual texturing system [van Waveren 09].

Extending this technique to arbitrary geometries and texture coordinate mappings adds a substantial overhead to the technique. Besides the streaming, (de)compression and texture updates, the additional cost of determining what parts of the texture are referenced by the rendered frame is introduced. Existing systems usually involve a lot of expensive CPU work and CPU to GPU data transfers.

In this chapter, we want to demonstrate how NVIDIA's compute unified device architecture platform can be used to reduce this CPU work and how it can be used to efficiently stream data between system memory and GPU memory. CUDA provides a straightforward way to address the GPU for general purpose computations, without the need to translate the problem in terms of shaders and textures. Although we implemented our system using CUDA, the theory will be equally applicable to upcoming vendor independent standards such as OpenCL or the DirectX Compute Shaders. The required hardware is getting cheaper and is no longer only available to the high-end market segment, making its use in commercial games attractive.

2.1 Introduction

2.1.1 Virtual Texturing

Virtual texturing is loosely based on the idea of virtual memory. The texture address space is logically divided into chunks called pages, then at runtime only the pages needed by the current view are loaded into fast texture memory. The rest of the texture is stored on disk in a page file and loaded by the game on demand. The set of pages needed by the current frame is referred to as the working set.

A major difference with traditional virtual memory is the presence of mipmaps. Distant portions of the texture should be loaded at a lower resolution. Mipmapping not only helps to reduce the working set: it also provides high quality, alias-free filtering of the texture in the distance. Another difference with traditional virtual memory is that we do not wait for a page to become available

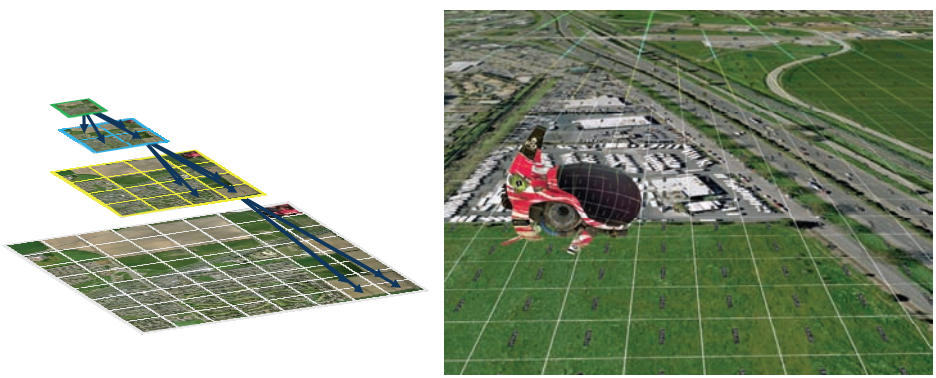

Figure 2.1. The logical page quad tree, and its application onto a three-dimensional surface.

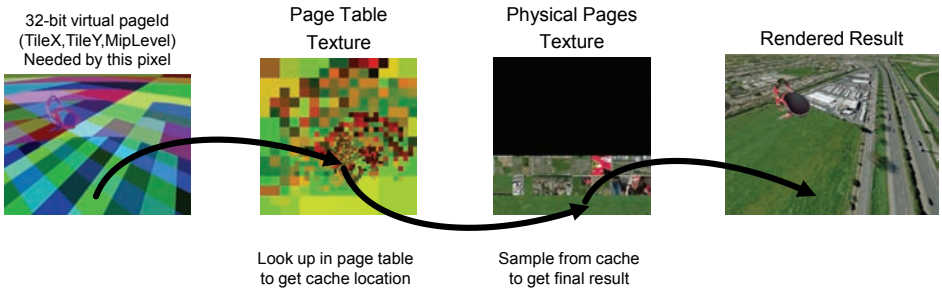

Figure 2.2. Schematic overview of the steps needed to render a virtual texture.

when a page miss occurs. Instead the system uses a lower resolution mipmap of the required page. This helps to ensure a constant frame rate. If the page loading can keep up with the view changes, this lower resolution fall back will generally not be noticeable to the player.

Logically the whole texture can be seen as a quad tree of tiles. At the root it is just a single page containing the lowest mipmap level. Every page then has four child pages on the higher resolution mipmap level. Figure 2.1 shows this page data quad tree structure and how it can be applied to a scene in three-dimensional. Notice how further away geometry references lower resolution mipmapped pages.

As with virtual memory, a page translation table (stored in a texture on the GPU) is then used to translate between virtual page identifiers (pageIds) and the location of the page in physical memory (a big cache texture allocated on the GPU). Figure 2.2 shows how these different data structures work together to get the final textured result.

The final result of rendering with virtual texturing looks like any traditional texture. In particular, there are no limits or requirements on the geometry or texture coordinates. Things like mirrored texture coordinates are transparently handled by the virtual texture system. From an artist's point of view, the system just "works," simply providing very detailed textures all over the world.

2.1.2 GPU Computing and CUDA

GPU computing has evolved a lot in recent years. It is moving away from graphics language based programming (e.g., Cg, HLSL, GLSL, etc.) to specialized parallel computing APIs. One of the first such APIs was NVIDIA's CUDA. CUDA provides a flexible programming model for mapping data parallel applications to the GPU. Currently the CUDA architecture can be accessed trough NVIDIA's proprietary extension to the C language. These extensions allow the programmer to specify the location where code and data needs to be located and executed.

This way, the programmer can transparently interface GPU code from CPU code without manually needing to manage the GPU. In the near future new ways to approach the GPU will become available. Both OpenCL and DirectX compute shaders will provide the programmer with a vendor independent GPU computing API with capabilities similar to C for CUDA. Porting to OpenCL from CUDA is reported to be relatively straightforward [NVIDIA 09].

Although the computing environment runs on the same hardware as the traditional shaders of the graphics pipeline, there are some important differences between these two environments. The first difference is the possibility to do scattered writes. While a graphics shader always generates a limited number of outputs in a set of predefined output registers, GPU computing environments allow arbitrary GPU memory access. This coupled with the ability to synchronize between different threads running on the GPU, allows more flexibility in the code and more effective parallel programming methods to be implemented. In addition to this, recent hardware also supports global atomic operations, making buffer and queue management easier. Finally, CUDA also provides asynchronous versions of its API. This allows the CPU and GPU to work independently of each other without needing unnecessary synchronization between them.

To efficiently use the power of these GPU computing environments it is still necessary to develop sufficiently parallel algorithms that work efficiently with the underlying SIMD hardware implementation. This means that diverging branches still imply a performance loss and that a sufficiently large number of threads has to be provided at once.

2.2 Implementing Virtual Texturing

To understand the rest of this chapter, it is necessary to provide some additional details about our implementation of virtual texturing. The architecture of our virtual texturing system is shown in Figure 2.3. The subsystems shown on the figure are largely independent in the code allowing us to easily test different approaches and acceleration strategies. We now briefly describe the function of the different subsystems and their interactions:

- The *page file* contains the source data for every page in our virtual address space. Mipmaps and pages are generated off line and compressed using a custom DCT (i.e., JPEG like) image compressor. Pages in our system contain a 120×120 pixel payload with a four pixel border on all sides. This ensures artifact free anisotropic filtering, which results in our textures getting sizes of a power of two multiplied by 120.

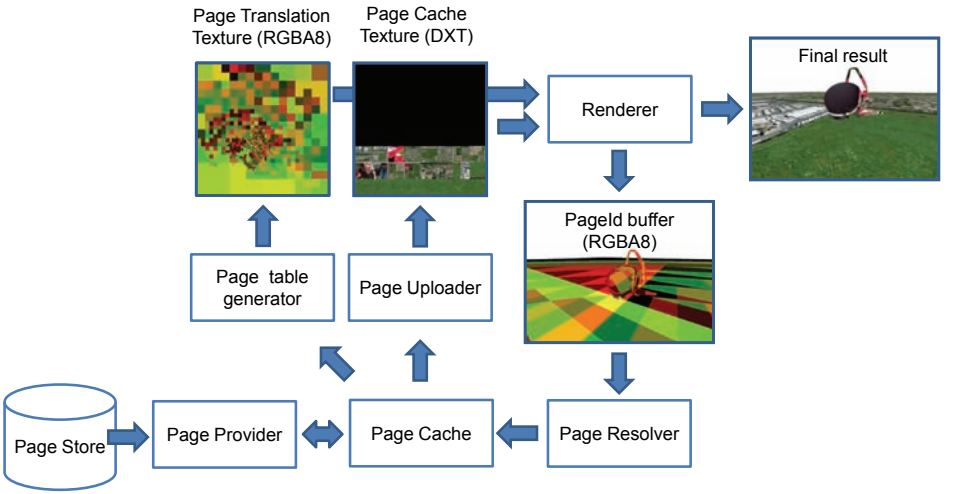

Figure 2.3. Overview of our virtual texturing system.

- The *page provider* provides the rest of our virtual texturing system with decompressed pages from the page file. The file loading and DCT decompression runs in a separate CPU thread and communicates with the main rendering thread via append/consume buffers.

- The main function of the *page cache* is to manage the physical texture. It uses a LRU strategy to decide what pages to replace when new pages are requested. The size of our cache is currently 4096×4096 texels (1024 pages). This is sufficient for rendering at a resolution of 1280×1024 with anisotropic filtering. We currently do not have special handling when the working set exceeds the cache capacity (the cache thrashes constantly). This could easily be handled by dynamically adapting the lod bias, resulting in gradually requesting lower resolution mipmaps till the full working set fits in the cache [van Waveren 09].

- The *page uploader* takes the loaded pages from the provider and efficiently transfers them to the GPU. On the GPU, mipmaps are generated and the results are encoded to the DXT format before being stored in the physical page texture. Because we want to use hardware accelerated trilinear filtering, our system requires the page cache to have mipmaps for its pages. By carefully sampling from the physical texture (see Section 2.3.2), it is sufficient for a single mipmap level to be present in order to support full trilinear filtering. In Section 2.4.2 we will describe these steps in more detail.

- The *page table generator* is responsible for maintaining the page table based on the list of pages present in the cache. Note that even a single page change could require updating large sections of the page table since pages on higher mipmap levels can span large portions of the virtual address space. We will show how we efficiently maintain the page table using the geometry shader in Section 2.4.3.

- The *page resolver* is responsible for determining which pages of the texture are needed by the frame. Our system uses an approach similar to [Barrett 08, van Waveren 08]. This works by rendering the view to a separate buffer containing the pageIds needed by the corresponding pixels. This buffer is then analyzed to extract the list of pageIds required by this frame. In Section 2.4.1 we will describe how this subsystem can be implemented on the GPU. The resulting list of pages (4kb of data) is then transferred to the CPU asynchronously and presented to the page cache.

- The *renderer* finally uses the textures prepared by the other subsystems to render the final texture mapped and filtered result.

2.3 Rendering with Virtual Texturing

In this section we describe how a virtual texture can be sampled and filtered assuming that the cache and translation textures have been correctly initialized. We will also show how to achieve trilinear and anisotropic filtering trough reuse of the existing graphics hardware. The shader we present in this section is mainly written for readability. It also uses the integer instructions available on the most recent generation of hardware. Integer instructions may, however, be slower than using only floating-point operations.

2.3.1 Translating Virtual into Physical Addresses

The translation texture contains for every virtual page the corresponding physical page. By using nearest filtering when sampling this texture, we can simply use the unmodified virtual texture coordinates (in the range [0-1]) to get the information about the required page. To make sure that the correct mipmap level of the page translation texture will be sampled we also scale the derivatives of the texture coordinates accordingly. If we do not factor in this scale, the hardware mipmapping unit would choose the mipmap level to achieve a 1:1 pixel ratio of the translation texture instead of a 1:1 ratio of the virtual texture. Since our virtual texture is larger than the translation table by a factor of the page size, we arrive at the following shader code:

```
float2 virDdX = ddx(i.uv)*PageContentSize;
float2 virDdY = ddy(i.uv)*PageContentSize;
float4 pageInf = tex2D(page_image,i.uv,virDdX,virDdY)*255;
```

The page translation texture then provides us with the following information: the X and Y channel contain the cache tile index, the Z channel the mipmap level of the cache tile, and the W channel the mipmap level of the virtual page. The first three parameters allow us to correctly scale the texture coordinates for sampling from the page cache. The fourth allows us to generate the pageId buffer without much additional work since we immediately know the mipmap level needed for this pixel. We will describe the pageId buffer generation in more detail in Section 2.3.4.

Finally, note that when using trilinear filtering, we need to set an additional mip bias of -0.5 on the page translation texture object. This ensures that the correct miplevels are requested by our system. A downside of this is that the frame's working set also increases since the range of texture data needed at the highest quality increases as a result of this bias. This effect will be even stronger when we use anisotropic filtering since this can require biases of up to the maximum degree of anisotropy.

2.3.2 Sampling the Cache Texture

We now have all the information to sample from the page cache. The page cache texture contains the physical pages, and the first mipmap level of the cache texture contains the page mipmaps generated by our uploading pipeline (see Section 2.4.2). Because we previously requested the lowest (i.e., highest resolution) mipmap level of the virtual texture we need to sample for this pixel. Mipmap levels beyond the second level will not be needed since trilinear filtering only blends between two levels in the mipmap. In theory these lower mipmaps could be sampled under extreme minification (e.g., when we need to sample the 4×4 pixel mipmap level of the whole $16k \times 16k$ texture). When these mipmap levels are needed we simply handle them by clamping to first mipmap level of the 1×1 page level of the virtual texture (i.e., we sample from the 60×60 mipped page where a 4×4 would be required). This could in theory lead to aliasing in this case. However, we never encountered any problems with this approach since it is unlikely to be so far away from the virtual texture that its screen space is less than 60×60 pixels.

We can now easily calculate the offset within this page by first calculating the offset in a [0-1] range and then converting this offset to pixels by scaling. We also add in the bias to account for the four pixel overlap between pages:

```
float2 cacheId = pageInf.xy;
float availableMipLevel = pageInf.z;
float numPagesOnLevel  = PagesOnAxis
      * pow(0.5,availableMipLevel);
```

```
float2 offset = frac(i.uv * numPagesOnMipLevel)
      * PageContentSize + BorderSize;
```

To calculate the updated derivatives for trilinear filtering, we first convert the input texture coordinates in the [0-1] range to pixel units on the current mipmap level. We then divide this value by the page cache size in pixels to convert those derivatives into the [0-1] space expected by the hardware when sampling from the page cache:

```
float deltaScale = numPagesOnLevel
      * PageContentSize * (1.0f / CacheSizePixels);
float2 sampDeltaX = ddx(i.uv)*deltaScale;
float2 sampDeltaY = ddy(i.uv)*deltaScale;
```

The final sample from the cache can be implemnted as follows:

```
float2 cachePos = cacheId * PageSize;
float4 final = tex2D(cache_image,
    (cachePos + offset)*(1.0f / CacheSizePixels),
    sampDeltaX,sampDeltaY);
```

2.3.3 Anisotropic Filtering

When working with anisotropic filtering the hardware automatically adds an additional negative bias to the mipmap level calculation. This bias depends on the degree of anisotropy. If the texture minification is near isotropic this bias is zero, if the texture is seen at a highly anisotropic angle this bias is clamped to the maximum level of anisotropy selected. This negative bias could of course lead to aliasing on the main axis of anisotropy. It is exactly this aliasing that is then avoided by doing multiple samples on this axis from the selected mipmap level.

If we want to ensure correct texturing results under anisotropic filtering we have to provide a similar bias to the mipmap level we select from our virtual texture. This bias can be calculated as follows. We refer to the anisotropic texture filtering extension specification [NVIDIA 99] for more details on this formula:

```
float deltaX = length(ddx(i.uv)*TextureSize);
float deltaY = length(ddy(i.uv)*TextureSize);

float deltaMax = max(deltaX, deltaY);
float deltaMin = min(deltaX, deltaY);
float N = min(ceil(deltaMax/deltaMin),MAX_ANISOTROPY);

int level = min(max((int)(log2(deltaMax/N)),0),MaxMipLevel);
```

Note that we cannot simply let the hardware select the mipmap level anymore when sampling from the page translation texture, since the hardware would not take into account the additional anisotropic bias we mentioned above. Trying to enable anisotropic filtering on the page translation texture to work around this would of course sample invalid blended pageIds. Hence, we have to explicitly provide the mipmap level that we request of our virtual texturing system:

```
float4 cache = tex2Dlod(page_image,float4(i.uv,0.0,level));
```

By simply sampling as described in Section2.3.2 and setting the anisotropy on the page cache as follows, we will then get exact anisotropic filtered results.

```
cacheTexture->bind();
glTexParameteri(GL_TEXTURE_2D,
                GL_TEXTURE_MAX_ANISOTROPY_EXT, 4);
```

2.3.4 Generating the PageId Buffer for the Resolver

As mentioned in Section 2.3.1, our translation texture contains the mipmap level of the page in its w channel. Hence by simply sampling from the page translation texture with the correct derivatives we can easily determine the mipmap level needed by this pixel. Once the level is known, the page indexes can easily be calculated based on the textures coordinates as follows:

```
int          levelSize = TextureSize >> mipLevel;
float2 virtualPixelsUv = i.uv * levelSize;
int2    virtualTilesUv = virtualPixelsUv * pixelsToPage;

int4 resultI =  int4(
  (virtualTilesUv.x & 0xFF),
  (virtualTilesUv.y & 0xFF),
  ((virtualTilesUv.x >> 8) << 4) | (virtualTilesUv.y >> 8),
  level
);
```

2.4 GPU-Based Acceleration

Many of the processes introduced in Section 2.2 lend themselves well to parallelization on the GPU. They involve pixel-like operations that can easily be parallelized. Secondly, much of the output data will be needed on the GPU for rendering so it is desirable to try and do the calculations as close to the GPU as possible. In the following sections, we will describe how we mapped some of the virtual texturing processes to the GPU.

2.4.1 Accelerating the Resolver

One of the biggest differences between a traditional texture streaming system and virtual texturing is the need for a resolver. Determining every frame which texture pages are needed for all the visible geometry in the frame is one of the most expensive steps in a virtual texturing system. The resolver usually works by first rendering a lower resolution view with a special shader that outputs the pageId needed by every rendered pixel followed by an analysis phase on the CPU that looks at the pageId's in the framebuffer and requests the new pages from the streaming thread (see Listing 2.1).

Our implementation differs from the previously described approach in several key aspects. First it does not require an extra rendering pass to generate the pageId buffer. This is achieved by outputting to an additional render target when the scene geometry is rendered. In a deferred rendering system this would ideally be during the G-buffer construction phase. In a more traditional rendering approach this could be done when rendering the z-prepass.

The downside of this approach is that our pageId buffer is generated at full resolution. However, rendering the frame geometry again is usually a lot more expensive than the additional shader and ROP cost of rendering the pageId buffer at full resolution. Because we will analyze the pageId buffer on the GPU there is also no need to stream the data to the CPU for the analysis step. So the increased size of the pageId buffer does not put an additional strain on the system bus. In fact, as we will describe below how our system actually reduces the CPU-GPU transfer by only transferring the compact list of required pageIds back to the CPU.

After rendering the frame, the prepared pageId buffer is then mapped in our GPU computing environment. Depending on the hardware capabilities and GPU computing API this might simply mean locking the graphics API's render target memory or doing a copy of the buffer to prepare it for processing by the GPU computing environment. One important note when using OpenGL through the Pixel Buffer Object extension is to use the BGRA pixel format. Data in this format can generally be copied faster because the hardware natively renders to BGRA and thus no byte swapping will be needed during the copy.

Once the buffer has been mapped to the GPU computing environment, a kernel is started that processes every pixel in the image and marks the pageId corresponding to that pixel as used. Instead of analyzing all the pixels in the buffer, we could reduce this cost by only starting a thread every nth pixel. During tests, we noticed no significant performance increase when analyzing the frame at a lower resolution.

After all the pageIds needed by this frame have been identified, a second kernel is started that packs the list of used pages to a single continuous buffer of page IDs. This buffer is then transferred to the CPU. To pack this list, we could use a

```
__global__ void markUsedPagesKernel(
    int *pixelBuffer, int width, int height,
    int frameId, int *outputBuffer)
{
    int2 pixelCoord;
    pixelCoord.x = blockIdx.x * blockDim.x + threadIdx.x;
    pixelCoord.y = blockIdx.y * blockDim.y + threadIdx.y;

    if ( pixelCoord.x >= width || pixelCoord.y >= height ) {
        return;
    }

    int4 pixel;
    pixel = tex2D(renderTexture,pixelCoord.x, pixelCoord.y);

    //Swizzle around (caused by BGRA rendering).
    int tileX = pixel.z;
    int tileY = pixel.y;
    int level = pixel.w;

    // Do some sanity checks on the shader output...
    if ( level > info.numLevels ) {
        return;
    }

    int levelWidth = sizeForMipLevel(level);

    if ( tileX >= levelWidth || tileY >= levelWidth ) {
        return;
    }

    // Calculate the level buffer.
    int *levelData = outputBuffer + offsetForMipLevel(level);

    // Mark this page as touched.
    levelData[tileY * levelWidth + tileX] = frameId;
}
```

Listing 2.1. CUDA kernel that marks the pages used by this frame.

prefix-sum based system [Harris et al. 08], but we instead opted to use a much simpler global atomics based system. This system works by maintaining a single counter that contains the number of pageIds required so far. When a thread encounters a required pageId, it increases the counter through an atomic operation to allocate a slot in the output list. It then continues to write the pageId to the allocated slot. Because the atomic operations are guaranteed to be mutually exclusive, no two threads can reserve the same output slot.

```
__global__ void gatherUsedPagesKernel(
    int *usedPages, int numPages,
    int frameId, unsigned int *outList)
{
    int offset = blockIdx.x * blockDim.x + threadIdx.x;

    // Check in range.
    if (offset > numPages) {
        return;
    }

    // A large portion of threads will return here.
    if ( usedPages[offset] != frameId ) {
        return;
    }

    int level = mipLevelForOffset(offset);
    int levelOfs = offset - offsetForMipLevel(level);
    int size = sizeForMipLevel(level);

    int x = levelOfs & (size-1);
    int y = levelOfs / size;

    // This will wrap around if more than MAX_FRAME_PAGES
    // are requested.
    int outIndex = atomicInc(outList,MAX_FRAME_PAGES);
    outList[outIndex+1] = make_pageId(x,y,level);
}
```

Listing 2.2. Cuda kernel to pack the list of used pages.

Since atomic operations require serialization of the memory accesses, a bottleneck may develop if many threads try to access the same counter. However, only a small number of pages is actually used every frame and hence there is a clear upper limit to the number of calls done on our atomic counter. Secondly, from performance measurements using the CUDA visual profiler, we determined that this step is certainly not the bottleneck. Hence, we decided use the current approach. The code of the packing kernel can be found in Listing 2.2.

The final step of our resolver is then transferring the packed list back to the CPU (see Figure 2.4). To ensure that we do not need to synchronize with the GPU, this transfer is started asynchronously. Because the actual number of pages that was emitted by the packer is not yet known to the CPU when the transfer is started, we instead transfer a single fixed size buffer containing the atomic counter and the maximum number of pages packed per frame. This results in more data transferred than strictly needed. However, it is much faster than starting

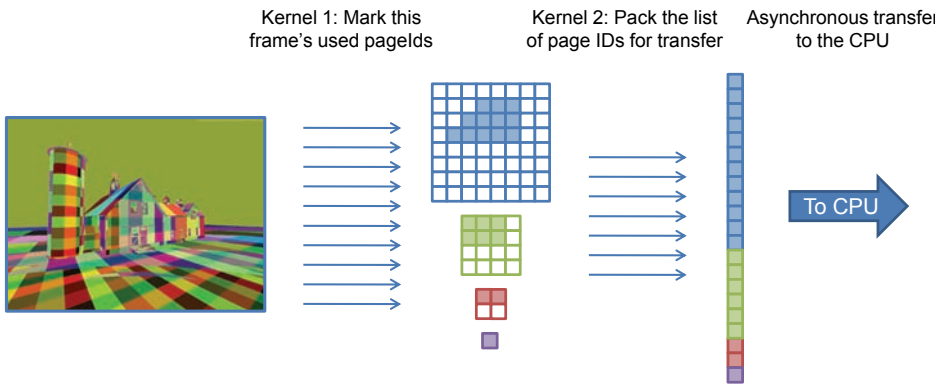

Figure 2.4. The resolver pipeline.

a synchronous GPU-CPU transfer to read back the number of pages emitted followed by an asynchronous copy of the actual page data. Also note that this buffer is only four kilobytes. Thus the overhead of starting a transfer generally overshadows the actual data transfer time.

2.4.2 Asynchronous Uploading and Compression

Another big issue with virtual texturing systems is getting the dynamically loaded data efficiently onto the GPU. For page data that needs no further processing, the Pixel Buffer Object(PBO) OpenGL extension [Biermann et al. 04] offers an efficient transfer path. When properly used, this extension provides high speed asynchronous data transfers between the CPU and GPU. Listing 2.3 shows how the PBO extension can be used to upload DXT compressed pages.

The `glBufferData` call at line two hints to the driver that we will replace the whole buffer. This will allow the driver to optimize our transfer knowing that the old data will not be referenced anymore. (The `WRITE_ONLY` parameter alone is not enough since it still allows us rewrite only certain parts of the buffer.) We will use a similar approach with VBOs in Section 2.4.3 to transfer the list of pages in the cache to the GPU when updating the page table.

PBOs allow only limited flexibility on the upload path and we would like to dynamically generate mipmaps and transcode our data to DXT. Hence, we chose to use CUDA for uploading and processing the page data. CUDA allows high-speed asynchronous copies from system memory to GPU memory. These asynchronous copies require the data to be in page-locked memory which can be allocated trough the CUDA API. To minimize the CPU processing of the uploaded data even further, our loading thread immediately loads and decodes the data to

```
glBindBuffer(GL_PIXEL_UNPACK_BUFFER_ARB, uploadFBO);
glBufferData(GL_PIXEL_UNPACK_BUFFER_ARB, size, 0,
   GL_STREAM_DRAW_ARB);
buff = glMapBuffer(GL_PIXEL_UNPACK_BUFFER_ARB,GL_WRITE_ONLY);
[put some data in buff...]
glUnmapBuffer(GL_PIXEL_UNPACK_BUFFER_ARB);
glBindBuffer(GL_PIXEL_UNPACK_BUFFER_ARB, copyFBO);
glBindTexture(GL_TEXTURE_2D, theTexture);
glCompressedTexSubImage2D(GL_TEXTURE_2D,
   0,xoffsets[idx],yoffsets[idx],
   PAGE_SIZE,PAGE_SIZE,
   GL_COMPRESSED_RGBA_S3TC_DXT5_EXT,
   DXT_PAGE_SIZE,
   (unsigned char *)(idx*DXT_PAGE_SIZE));
```

Listing 2.3. Pseudocode to upload a page using the PBO OpenGL extension.

page locked memory. This way, the CPU does not need to touch the data outside of the page provider thread.

When new pages are made available by the page provider, our main thread simply passes the buffer to CUDA to be uploaded asynchronously. It then calls the necessary kernels to generate the mipmaps and to do the DXT encoding. The DXT encoder immediately writes its results to an OpenGL Pixel Buffer Object. Finally, the buffer is then transferred to the page cache texture using high-speed GPU-GPU memory transfers.

Modern GPU hardware allows overlapping kernel execution with data transfers. We exploit this capability by starting to encode the first few pages while additional pages are still being downloaded. To use this capability, we have to organize our CUDA calls in two "streams." All CUDA calls executed within a single stream are executed sequentially. The GPU will not start executing our encoding kernel before the asynchronous data transfer has finished. However, calls in different streams are not ordered across stream boundaries; thus, by executing our uploading and compression in several streams, parallel uploading and encoding of pages can be achieved.

A second thing to take into consideration is that CUDA kernels only perform optimally if they are given enough work (i.e., threads) to complete in one time. Ideally, several thousands of threads have to be provided to the hardware at once. Because we perform the DXT encoding with a single thread per 4×4 block, we have only 1024 threads available per page (i.e., 32×32 DXT blocks on a 128×128 pixel page). This is too low to efficiently use the full capacity of the graphics hardware. Therefore we first upload several pages before starting the encoding process on a group of pages. Although our demo only loads diffuse textures additional channels of texture data, such as normals and specular colors, should also help to

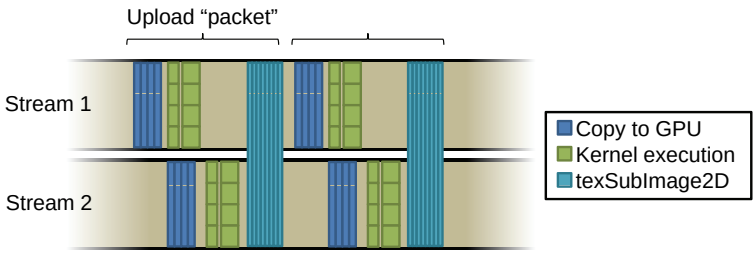

Figure 2.5. Parallel uploading and compression of upload packets using CUDA streams.

increase the thread count to a level that optimally uses the available computing power.

Finally, we would like to have a limit to the number of GPU buffers used for the compression and uploading of the pages. Therefore we split the data into “upload packets.” A fixed number of packets is uploaded, compressed and copied to the texture. In this way we only need buffer space for the pages in a single packet. This packeting also helps to reduce the switching overhead between the GPU computing and graphics contexts (i.e., CUDA and OpenGL), which may be an expensive operation depending on the platform.

Figure 2.5 gives an overview of how the pages are grouped in packets and then split over several streams to be uploaded and compressed. After mipmap generation and compression, the streams are again synchronized and the data is transferred to the cache texture using `texSubImage2D`.

We now discuss the actual kernel implementation of the upload system. The first kernel in our system is the mipmap generation. Mipmaps are currently generated by a simple 2×2 average. The memory where the pages were uploaded is first mapped as a two-dimensional texture. Mapping memory as textures does not require and additional memory copy under the latest versions of CUDA. Reading the uploaded data trough a CUDA texture has several advantages compared with just reading them as raw memory. First CUDA textures are cached with optimizations for two-dimensional locality. Although our kernel only reads the input data once, this may still help since the data of neighboring threads can be fetched into the cache together. Secondly, texturing allows automatic unpacking of the data without requiring manual shifting and masking of the RGBA8 data. Finally, CUDA textures like their OpenGL counterparts allow for hardware accelerated filtering. Filtering allows us to do the 2×2 average with a single texture sampling operation.

The second kernel then compresses the uploaded data to the DXT texture compression format. This kernel is currently invoked twice, once for the base level and once for the generated mipmaps. There has been much previous work on

achieving real time DXT compression [van Waveren 06]. Although these algorithms were developed for CPU use, they are also efficient on GPUs since they do not cause a lot of diverging branches. We again sample the source data through a texture since we require a lot of spatially local samples to encode a 4×4 block.

The most significant disadvantage of our current implementation is that it requires a lot of registers while processing the 16 pixels in a block sequentially. This means that the GPU can schedule less threads at once, preventing it from optimally hiding memory access latencies. Modifying the code to use fast per SIMD-unit shared memory besides registers is not an ideal solution either as this memory is equally limited in size. The best solution would be to move away from having one thread per 4×4 DXT block toward a system that uses one thread per pixel or per channel. These approaches would lead to 16 or 4 threads per DXT block with a significantly reduced per thread register count. This leads to more threads but also more calculations running in parallel with less sequential processing per block. These modifications will likely result in improved compression times.

2.4.3 Updating the Page Table

As described in Section 2.2, the page table provides the translation between a virtual page identifier and an actual physical page. Since not all requested virtual pages are necessarily available in the cache, the page table contains the best available physical page for every virtual page. It also needs to provide the mipmap level of that physical page so that the texture coordinates within the page can properly be adapted.

This page table can easily be generated on the CPU and then transferred to the GPU. By carefully maintaining the modified regions a small number of texSubImage 2D calls suffice to update the page table.

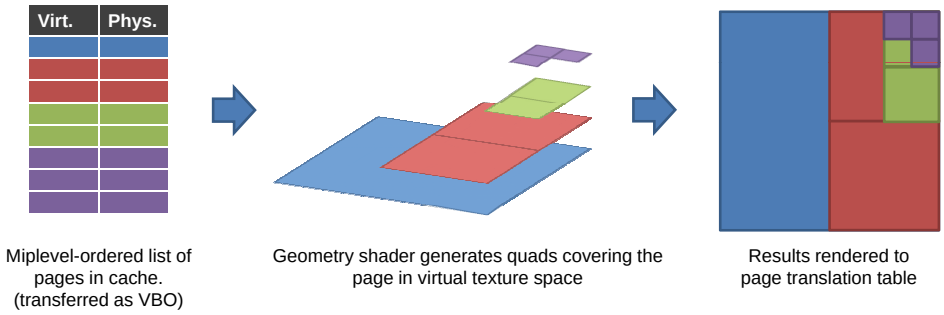

Figure 2.6. Generating the page translation table using a geometry shader.

```
POINT TRIANGLE_OUT
void geometry_main(AttribArray<vsVertex> inverts) {
    float4 pageId = inverts[0].pageId*255;
    float2 physId = inverts[0].physId;
    int currentLevel = (int)LevelInfo.x;
    float levelScale = LevelInfo.y;

    // Higher miplevels cover larger areas
    // in the translation table.
    float scale = (1<<((int)pageId.w-currentLevel));
    pageId.xy *= scale;

    gs2psVertex vert;
    //Write the physical page address and the number of pages
    //on the miplevel of this virtual page. We write this
    //instead of the miplevel because it saves operations
    //in the pixel shader.
    vert.color = float4(physId.x/255.0,physId.y/255.0,
                0.0,((int)PagesOnAxis>>pageId.w)/255.0);

    //Generate a quad & range compress to normalized dev coords.
    vert.pos = float4(float2(pageId.x      ,pageId.y      )
          *levelScale*2-1,0.0,1.0);
    emitVertex(vert);
    vert.pos = float4(float2(pageId.x+scale,pageId.y      )
          *levelScale*2-1,0.0,1.0);
    emitVertex(vert);
    vert.pos = float4(float2(pageId.x      ,pageId.y+scale)
          *levelScale*2-1,0.0,1.0);
    emitVertex(vert);
    vert.pos = float4(float2(pageId.x+scale,pageId.y+scale)
          *levelScale*2-1,0.0,1.0);
    emitVertex(vert);
}
```

Listing 2.4. Cg code for the page table geometry shader.

To simplify the system and to guarantee a constant and minimum amount of data transfer between the CPU and the GPU, we chose to implement the page translation texture generation on the GPU as well. We transfer information about pages currently available in the physical cache to the GPU and then use a geometry shader while rendering to the page table texture mipmap levels to generate the page table. In our current implementation this results in a transfer of 12 bytes per page (or 12 kilobyte for the whole translation table).

Figure 2.6 shows how the geometry shader reads the physical page information we transferred and generates a quad of the correct size in virtual texture space to render into the translation texture. By ordering the page table information we send from lowest to highest resolution, we avoid needing a depth buffer when

rendering to the page table texture. This ordering only takes a minimal amount of extra work at the CPU side since we simply maintain a linked list per mipmap level without requiring any sorting on the CPU. The code of the geometry shader can be found in Listing 2.4.

2.5 Results

We have now finished presenting the various GPU-based optimizations our virtual texturing implementation provides. In this section we will briefly discuss the performance of our system. These results should give a clear picture of the performance impact of our technique on traditional game rendering architectures. Table 2.1 shows the main performance results for our GPU-based virtual texturing system. These results were measured on an Intel Core 2 Quad CPU, with a NVIDIA GeForce GTX 285 graphics card and 2 gigabyte of RAM. The results are expressed in milliseconds per frame so it should be easy to determine how much the different virtual texturing subsystems will contribute to a game frame. All results where averaged over 100 frames. The page upload time is also averaged after uploading a full upload packet as described in Section 2.4.2. Note that the page uploading and page table updates do not necessarily have to be done every frame. These tasks will only be executed when new pages arrive from the background loading thread. We did not include results for the CPU based steps of our loading thread since we did not do any special optimizations on them. However, they are currently capable of providing pages on time without any visible popping.

From this table we can conclude that we can compress, generate mipmaps and update the cache texture for more than 4,700 pages per second. This is equivalent to about 78 mega pixels per second. Several times more than needed to update small portions of the cache per frame. In frames per second our system performs well over 400Hz with a static camera and around 350Hz when moving around at walking speeds.

Comparing our results with other systems is difficult since not many results from other sources are available. One source [van Waveren 09] quotes 8ms of virtual texturing overhead per frame. Although details are not mentioned, we can conclude that our system should at least perform equally well if not better under similar circumstances.

Subsystem	Frame times
Resolver	1.2ms
Update page table	0.7ms
Upload+Process 1 page	0.21ms

Table 2.1. Performance results of the virtual-texturing subsystems.

2.6 Conclusion

In this chapter we have shown how CUDA and GPU computing in general can benefit virtual texturing and game rendering. From our results we can conclude that GPU-based virtual texturing is a promising game technology that offers high real-time performance. Although it creates extra GPU overhead compared with normal texturing, it also helps to make other areas of the renderer faster. Texture blending, decals, batches and extra geometry to support texture atlases can all be greatly reduced since the frame is rendered with only a few detailed textures. This, coupled with the simplicity of the texturing working all over the game world without any special care by artists, makes it very attractive for deployment in game engines. In the future we hope to investigate more efficient compressions systems and how they can be efficiently mapped to the GPU. This will allow us to keep more texture information in the virtual texture and should allow more advanced lighting models. In addition to this, we also want to investigate how this technique can be expanded to alpha blended textures, leading to a fully transparent virtual texturing system.

2.7 Acknowledgments

The research activities that have been described in this chapter were funded by Ghent University, the Interdisciplinary Institute for Broadband Technology (IBBT), the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT), the Fund for Scientific Research-Flanders (FWOFlanders), the Belgian Federal Science Policy Office (BFSPO), and the European Union.

Bibliography

[Barrett 08] Sean Barrett. "Sparse Virtual Textures." In *GDC 2008 presentation*, 2008.

[Biermann et al. 04] Ralf Biermann, Derek Cornish, Matt Craighead, Bill Licea-Kane, and Brian Paul. "Pixel Buffer Object."

[Harris et al. 08] Mark Harris, Shubhabrata Sengupta, and John D. Owens. "Parallel Prefix Sum (Scan) with CUDA." *GPU Gems 3*, pp. 851–876.

[Lefebre et al. 04] Sylvain Lefebre, Jérome Darbon, and Fabrice Neyret. "Unified Texture Management for Arbitrary Meshes." *Research Report No. 5210.*

[Mittring 08] Martin Mittring. "Advanced Virtual Texture Topics." *ACM SIGGRAPH 2008 Classes.*

[NVIDIA 99] NVIDIA. *Texture Filter Anisotropic*. Santa Clara: NVIDIA Corporation, 1999.

[NVIDIA 09] NVIDIA. *OpenCL JumpStart Guide*. Santa Clara: NVIDIA Corporation, 2009.

[Tanner et al. 98] Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones. "The Clipmap: A Virtual Mipmap." *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*.

[van Waveren 06] Jan-Paul van Waveren. "Real-Time DXT Compression."

[van Waveren 08] Jan-Paul van Waveren. "Geospatial Texture Streaming from Slow Storage Devices."

[van Waveren 09] Jan-Paul van Waveren. "ID Tech 5 Challenges." *SIGGRAPH 2009: Beyond Programmable Shading Course*.