Texture Caches

Michael Doggett
Department of Computer Science
Lund University
Sweden
Email: mike (at) cs.lth.se

This paper is a preprint of an article that appears in IEEE Micro 32(3), May-June 2012.

I. Introduction

The texture cache is an essential component of modern GPUs and plays an important role in achieving real-time performance when generating realistic images. The texture cache is a read-only cache that stores image data that is used for putting images onto triangles, a process called *texture mapping*. Figure 1 shows a typical real-time graphics scene rendered with and without texture mapping which adds color details to the triangle models that make up the 3D scene. The texture cache has a high hit rate since there is heavy reuse between neighboring pixels and is usually located close to the shader processor so texture data is available with high throughput and at low read latencies.

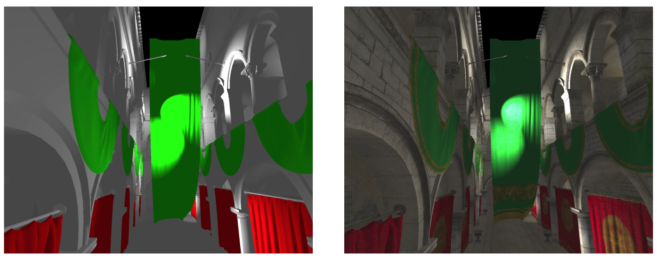

Fig. 1. The sponza atrium palace model on the left with colored triangles and lighting, on the right with texture mapping.

GPUs are large complex systems and the texture cache is only a small component. An overview of GPUs and their evolution is presented by Blythe [1]. Blythe points out that texture mapping was an early requirement for games and was one of the first features to move from high end research systems to PC and console gaming. The GPU has evolved in a competitive market that has seen most companies disappear from the many that existed in the late 1990s. The last major GPU vendor, NVIDIA, has expanded beyond just GPU design and also builds integrated GPU and CPU SOCs for the mobile market.

GPU architecture is not visible to the programmer and is hidden under the programming APIs (e.g. OpenGL and Direct3D) and so architecture, including texture caches, has changed frequently. A generalized top level architecture diagram of a modern GPU is shown in Figure 2. The texture cache, L1 cache and L2 cache make up the memory hierarchy in the GPU.

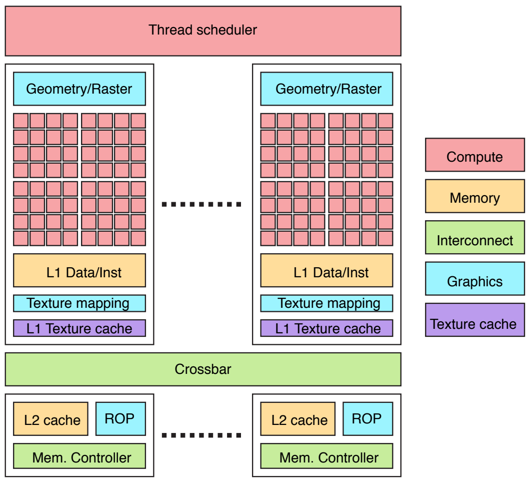

Fig. 2. A simplified diagram of the top level architecture of a modern GPU. The texture cache is highlighted in purple. The thread scheduler collects workloads for the different stages of the graphics pipeline and loads then onto a variable number of graphics cores. The graphics cores have some fixed function graphics logic in the Geometry/Raster unit, which contains geometry processing and rasterization. Underneath is the shader, it has control logic (not shown) and a grid of ALU units that process small programs in a SIMD fashion. The shader makes use of the L1 data and instruction caches, texture mapping unit and it's associated texture cache as the programs are executed. The output of the shader and the connection of the L1s to the L2s goes via a crossbar The crossbar is connected to a variable number of memory units, which contain the L2 cache, the ROP (fixed function graphics) and the memory controller, which is connected to the external memory where the framebuffer is stored. Most neighboring blocks are connected to each other, so interconnections between blocks are not shown.

To get more detail about GPU architecture is challenging. One of the most detailed lower level descriptions of how the graphics algorithms and units inside a GPU interact with memory, including the texture cache, is the description of the Neon GPU architecture by McCormack et al. [3]. But the technology used is quite dated, so some concepts are no longer relevant. For more modern GPUs, Wong et al. [4] use extensive benchmarking to determine the CUDA visible characteristics of NVIDIA's GT200 architecture. They determine characteristics of the processing elements and the memory hierarchy.

On the topic of texture cache design, there is also little literature. Hakura and Gupta's seminal paper [2] gives the best overview of the principles of a texture cache, including the effects of data locality, texture blocking, and tiled rasterization, all topics that we will revisit in this article.

To understand the requirements of a texture cache we first need to look at the units in a GPU that impact the cache functionality, that is it's client, texture mapping, and the unit that determines pixel ordering, rasterization.

II. Texture mapping

Texture mapping is the process of taking image data and placing it onto a surface. The image data is typically two dimensional (2D), but can also be one and three dimensional (1D, 3D). At each pixel on the screen the corresponding value has to be found in the image data, commonly known as the texture map. The individual elements in the texture map are called *texels* (from TEXture ELements) to differentiate them from the screen pixels. Each vertex of a triangle contains a 2D position on the texture map. These positions are called *texture coordinates* and are usually denoted by u, v. Figure 3 shows a pixel in a triangle using its u, v coordinate to look up the corresponding texel in the texture map which is in s, t coordinates. The texture map texels are in their frame of reference, s, t which ranges from 0 to 1 along the s and t axis. The u and v coordinates typically range from 0 up to the texture's width and height, but can also go beyond to allow effects such as texture wrapping. The texture coordinates are calculated by interpolating values at each vertex, and then the coordinates are used to look up the color which is applied to the current pixel. Looking up a single color based on the integer coordinates is called *nearest* filtering and results in aliasing. To avoid this aliasing, filtering of neighboring pixels is used. The first type of filtering is *bilinear*, in which linear interpolation of neighboring pixels in the X and Y dimensions is computed. This smooths out the texture mapping, but does not handle minification of the texture, where many texels should be averaged into the current pixel.

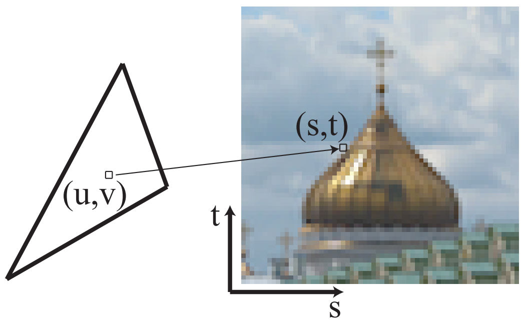

Fig. 3. Texture mapping involves finding the corresponding color in a 2D image map in s, t coordinates using the texture coordinates u, v at the current pixel in a triangle.

To handle minification a technique called *mipmapping* [5] is used. Mipmapping prefilters the texture map into a series of maps that are half the dimensions in X and Y. The sequence of maps can be stacked in a pyramid from the largest map at the bottom to the smallest map at the top. Figure 4 shows a mipmap stack with the highest resolution image at the bottom, level 0, and subsequent levels going up, each half the resolution of the previous level. Trilinear mipmapping works by finding two neighboring maps in the pyramid, and performing bilinear filtering on each map and then a linear interpolation between the two values, resulting in trilinear filtering. This can be seen in Figure 4 where level 0 and level 1 are selected and a neighborhood of 2×2 pixels are found for bilinear filtering on each level. To select the two neighboring maps, the Level Of Detail (LOD) must be calculated. The LOD is calculated by taking the difference in texture coordinates between neighboring pixels in a 2×2 grid. More complex filtering such as anisotropic filtering [6] is used in texture mapping, but it is typically done using a sequence of bilinear filtering operations making its requirements on the texture cache similar to bilinear filtering.

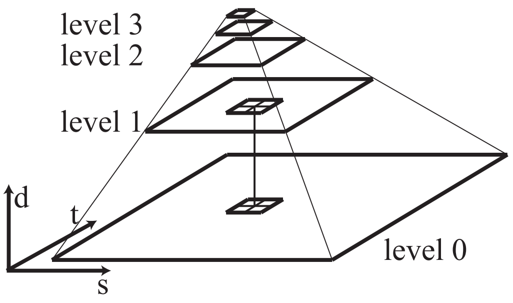

Fig. 4. A sequence of images making a mipmap pyramid. Level 0 is the full resolution image and each subsequent level is a half size averaged version of the previous level. Texture filtering first calculates the level of detail, d, which selects two levels of the pyramid and then performs trilinear filtering between the two layers.

III. Rasterization

Rasterization is the process of breaking up triangles into screen pixels. Triangles are the basic primitive used by GPUs, since by limiting the rasterization process to using triangles, the algorithm can be optimized for implementation in custom hardware. The simplest order in which to generate pixels from a triangle is to move down the edges of the triangle and scan across each horizontal line from one edge to the other. While this is straight forward, it is not the most optimal and so the order is changed to improve performance as presented in Section V.

GPU rasterization also groups pixels into 2×2 blocks of pixels, commonly referred to as a quad. Each pixel stores it's u, v coordinates and the texture gradients, $\delta u/\delta x, \delta v/\delta x, \delta u/\delta y, \delta v/\delta y$, are calculated by taking the difference of two neighboring pixel's texture coordinates. These gradients are then used to calculate the LOD as specified in the OpenGL specification [7]. If a triangle does not cover the entire quad, the GPU still pads out the quad with null pixels, which occupy space in the GPU compute engines. This leads to single pixel triangles being very inefficient for GPUs, but solutions to packing these pixels together, when they are part of a triangle strip/mesh, have been proposed by Fatahalian et al. [8].

More details on texture mapping and rasterization can be found in Akenine-Möller et al.'s book [9].

IV. Texture map memory

The order of texels accessed in a texture map can have any orientation with respect to the rasterization order. If the texels in a texture map are stored in memory in a simple linear row major ordering, and the texture is mapped horizontally, then cache lines will store long horizontal lines resulting in a high hit rate. But if the texture is mapped at a 90 degrees rotation, then every pixel will miss and require a new cacheline to be loaded. To avoid this orientation dependency textures are stored in memory in a tiled, or blocked [2] fashion. Each texture is broken into a series of $n \times n$ tiles and all the texels in the tile are stored before moving onto the next tile.

Hakura and Gupta [2] found that higher cache hit rates are achieved when the tile size is equal to cache line size for cache sizes of 128KB and 256KB and tile sizes of 8×8 and 16×16.

Another important aspect of memory usage is the concept of working set size. The working set size to render one frame is proportional to the resolution of the image according to 'The Principle of Texture Thrift' [10]. Hakura and Gupta [2] found that if the working set fits into the cache then miss rates are further reduced.

Texture caches are read only since the typical usage of a texture is for it to be placed onto a triangle. But computing the values that go into a texture map using the GPU has become common place and is called "render to texture". So even though a texture cache is typically read-only and the memory buffer in the GPU's main memory is tagged as read-only, it can also be changed to writeable. If a texture map's state is changed to writeable, all on-chip memory references to that texture must be invalidated and then the GPU can write to the texture. A simpler brute force method which flushes all caches in the GPU, can also be used. This invalidation of caches is required to ensure that the many specialized caches in the GPU, such as the texture cache, are maintained in a coherent state.

V. Tiled rasterization

Similarly to tiling for texture maps in memory, tiling also improves the access order of texture data when used for rasterization. A simple rasterization process could generate pixels in a row major order, starting at the left hand side of a triangle and generating pixels across the row to the right hand side. For very large triangles a texture cache could fill up as pixels are generated across a horizontal span and on returning to the next line find that the texels from the previous line have been evicted from the cache. This horizontal rasterization results in texture fetches that vary in terms of spatial locality across the width of the triangle. To improve this spatial locality, tiled rasterization is used instead. Tiled rasterization divides the screen into equally sized rectangles, typically powers of two, and completes the rasterization within each tile before moving to the next tile. The order of texture access can be further improved by using hierarchical orders such as the Hilbert curve [11] or Z-order.

VI. Texture cache architecture

The role of the texture cache has changed with the introduction of read/write L2 caches in GPUs such as NVIDIA's Fermi [12] and AMD's GCN [13]. Hakura and Gupta [2] presented single level, per pipeline units and considered cache sizes of 4KB, 32KB and 128KB. In terms of cache size, the modern GPU texture cache L1 is approaching a similar size, for example it is 12KB in the NVIDIA Fermi and 16KB for the AMD GCN. The texture cache changed into a two level cache as seen in the NVIDIA G80 architecture [14]. The G80 had an L2 cache distributed to each DRAM channel. The L2 is then connected to the texture L1 units via a large crossbar as shown in Figure 2. The per memory controller L2 is now a common GPU memory architecture design. By placing one unit of the L2 at the memory controller the GPU can be easily scaled in terms of memory controllers independent of the number of L1s and hence shader units. When designing the memory hierarchy, making this L2 cache a linear memory mapped cache is better than customizing it for texture. This allows the L2 cache to have more than just texture as a client and has made for a more straightforward transition to the modern read/write L2.

Starting from the top level GPU architecture in Figure 2, we see that the texture cache is located between the texture filtering unit and the crossbar. The internal architecture of a texture cache is shown in Figure 5. Boundaries between blocks in a GPU are arbitrary, so some of the operations could instead be placed in a texture filtering unit, but we show them here to get a better understanding of what could be required of a texture cache.

The texture cache is designed to connect well with the texture unit. The texture unit requires 4 texels in parallel in order to perform one bilinear operation, so most texture caches are designed to feed these values in parallel in order to maintain full rate for bilinear filtering. Most GPUs have multiple texture filtering units running in parallel, and the texture cache must supply these with texels. To do this the texture cache uses multiple read ports, or else replicates the data in the cache.

Texture filtering and texture cache units have to support every format that is offered by the graphics APIs. Which formats are supported for custom hardware texture filtering can vary greatly depending upon a GPU vendor's best estimate of what requirements are important for current and future games. The alternative is to support texture filtering in the shader at much lower performance. If a format is only supported by the texture cache, then it could be required to do the final formatting as the value does not go through the texture filtering unit. Texture formats can include bit widths from 1 to 32 bits and number formats including integer and floating point. If these formats are supported, then final formatting can be done by the texture cache instead of the texture filtering and the

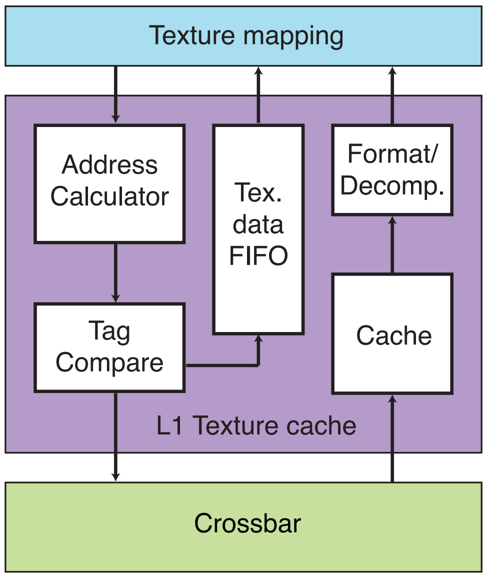

Fig. 5. An overview of a texture cache architecture. The texture mapping unit provides texture coordinates for which a memory address is calculated. The address is sent to the tag compare to determine if the data is in the cache. If the data isn't in the cache, a request is sent via the crossbar to the L2 cache. Any state associated with the original request is sent into a FIFO to return to the texture mapping unit with the texel data. Once the data arrives in the cache, or is already available in the cache, it is returned to the texture mapping unit. If the data is compressed, it is decompressed and any formatting that is required is done.

texture cache can include logic to shift bits and convert texels into the correct formats for the shader.

A. *Texture compression*

Texture compression [15], [16], [17] is an important component of ensuring texturing performance by reducing texture bandwidth. Since decompression of the texture happens somewhere between reading the compressed texture from memory and giving texels ready for texture filtering to the texture pipe, texture compression can sometimes be found within the texture cache design. Texture decompression is usually after the cache since storing compressed data in the cache means more texture data gets cached and the cache is much more effective. For example, the most popular texture formats S3TC [18] (DXTC in Direct3D) and ETC [19] (used in OpenGL ES), have a 6:1 compression ratio for RGB textures. Another important aspect of texture compression is that the logic must be small. This is necessary as wherever the logic is located it will be replicated many times making any added complexity very expensive. This has lead to a very slow change in texture compression formats with new ones only recently being added in Microsoft's Direct 3D 11.

VII. Performance

Texture cache design is heavily dictated by performance. It is important that a texture cache can sustain the full external memory bandwidth with texture data to the texture unit. An important design target for the texture unit is to be able to perform one bilinear filtering operation per clock cycle, which requires 4 texels. This means that the texture cache must be able to deliver 4×32 bits per clock to the texture unit. For an L1 cache with 4 texture unit clients it is 4 times that. Given a target hit rate determined from a cross section of current games, an input bandwidth to the L1 can be determined and a corresponding output bandwidth from the crossbar can be determined. This output must then be sustained by the L2 cache as well.

When a cache miss occurs and a new cache line must be read in from off-chip memory, a large latency is incurred. Igehy et al. [20] propose to hide this latency by precomputing the addresses required and pre-fetching the cachelines. Any associated texture filtering state that is needed when the texel data is returned to the texture filter is FIFOed while it waits for texel data to be returned from the cache. This means that texture accesses that hit in the cache must wait in the FIFO behind others that miss. A FIFO is used because the graphics pipeline ensures that triangle submission order is maintained, so allowing some texture fetches to complete early would not have a large impact on performance. FIFOing the texture requests leads to long latencies since modern GPUs have several hundred clock cycle latencies from the time a memory request is made in the shader pipe until data is returned and the shader can resume. These long latencies are hidden by the GPU running many thousands of threads [14], which enable the GPU to switch to other work immediately and ensure the computational units of the GPU are kept active while some threads wait for memory to load. These threads run on the shader with all register space allocated in shader memories, so that single cycle thread switching is possible ensuring no cycles are lost waiting for memory requests.

While the basic principles of cache performance presented by Hakura and Gupta [2] haven't changed, many of the parameters have changed dramatically. For example, they worked with a total texture size of between 1 and 56 MBs, compared to modern PC games which could easily use half of the available memory on PC cards, which range up to 3GB. Most major games are cross-platform titles and hence limited by the memory of the consoles, which is around 512MB. A game could easily use half of that memory for texture. This leads to higher resolution textures in modern games compared to what Hakura and Gupta worked with. At the extreme end, technology such as id Tech 5's virtual texturing is able to support $128,000 \times 128,000$, although this technology only uses a small amount of GPU memory for swapping small texture tiles in and out. This increase in texture storage leads to an increase in number, size and resolution of textures. Textures on mobile platforms have similar limits with high end smart phones also having 512MB of memory. But mobile platforms are more impacted by power usage and if a lot of texture memory is used, the power usage can be increased. In the future the usage of texture memory will follow the technology, as memory gets bigger, so will texture usage. For the mobile platforms a steady growth can be expected. But for the consoles a new generation is expected soon, and will bring with it larger memory space for textures, most likely enabling console games to have more texture than mobile games. This disparity in available texture memory, which can occur between different PC graphics cards, can be handled by using higher resolution textures when more memory is available.

The amount of repeated texture has decreased over recent years in games. The test scenes used by Hakura and Gupta had very large triangles in terms of pixel area (41, 294, 1149, 186 pixels). Most modern games have pixel areas per triangle of less than 10 pixels, driven in part by modern tessellation engines such as in NVIDIA's Fermi [12]. The current limit to game triangle size is the GPU's quad, which means that going below 4 pixels severely impacts GPU shader utilization.

Cache performance is also impacted by associativity. Hakura and Gupta showed that 2-way set associativity worked well to reduce cache line conflicts between mip-map layers, but modern applications often use many textures per pixel, and higher levels of associativity are required. For example Wong et al. [4] find the associativities of the GT200 to be 20-way for the L1 cache and 8-way for the L2 cache.

Modeling the texture cache and the GPU's memory hierarchy can help to better understand it's performance. While this is difficult, early work shows that this can be done for applications such as real-time ray tracing [21].

VIII. General Purpose GPU

In recent years, using GPUs for general purpose parallel computing has become common place [22], and is typically referred to as GPGPU. Texture mapping is part of the dedicated hardware designed for graphics rendering, but it can also be accessed in GPGPU languages such as OpenCL [23] by using the image and sampler objects. In modern read/write L2 cache GPU architectures, the texture cache doesn't offer a great advantage over the shader load/store L1 cache, except that it is an additional cache which is otherwise unused. The texture filtering custom hardware can be put to good use in GPGPU applications, if a good match for the filtering operations can be found.

GPGPU is pushing GPU architecture into a more general direction, as evidenced by the introduction of the read/write L2 cache. Other features which are not used by graphics are also improving in GPUs, such as atomics, 64-bit floating point performance and support for Error Correcting Codes (ECC). These features while desirable for GPGPU, take up chip area for graphics, making the GPU more expensive. A solution to this problem is to make two differently configured GPUs from the same architecture, one configured for graphics and one configured for GPGPU.

IX. GPU trends

The consistent economic conditions that existed in the first decade of the twenty-first century and drove the GPU from a simple graphics display adaptor into a massively parallel device, are not expected to continue in the second decade. With CPU and GPU integration becoming more mainstream, and with an increase in the graphics performance of portable devices, the dominant graphics device will be an integrated one. At the high end the GPU is driven by a small market for high end graphics and GPGPU. Today neither market is large enough to fund the research expense required for high end GPUs. The expense still requires the mid-range graphics market. So it has been risky to produce chips just for GPGPU, but this is starting to change with the continuing growth of the GPGPU market. The console market on the other hand takes a GPU design and keeps it constant over many years, typically around 5 years. This allows custom GPUs to be designed for a particular console without the need for all the GPGPU features. If the GPGPU market can sustain the development of a high end device without graphics, then these devices could be built without texture caches. If texturing performance could be made to be acceptably close to dedicated hardware, then it's possible that texture hardware and the associated texture cache could be removed from GPUs. But graphics performance is still an important factor for GPU sales and cannot be neglected while two strong competitors exist. So even though different markets require different memory architectures, GPUs will continue to be built for both markets and a texture cache will continue to be a necessary unit for high performance.

Acknowledgment

Thanks to Jocelyn Houle and Tomas Akenine-Möller for proof reading. I acknowledge support from the Intel Visual Computing Institute, Saarbrücken, Germany and the ELLIIT Excellence Center at Linköping-Lund in Information Technology. The Sponza Atrium Palace model is courtesy of Crytek.

References

[1] D. Blythe, "Rise of the graphics processor," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 761 –778, may 2008.

[2] Z. S. Hakura and A. Gupta, "The design and analysis of a cache architecture for texture mapping," in *Proceedings of the 24th annual international symposium on Computer architecture*, ser. ISCA '97. New York, NY, USA: ACM, 1997, pp. 108–120. [Online]. Available: http://doi.acm.org/10.1145/264107.264152

[3] J. McCormack, R. McNamara, C. Gianos, L. Seiler, N. P. Jouppi, and K. Correll, "Neon: a single-chip 3d workstation graphics accelerator," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, ser. HWWS '98. New York, NY, USA: ACM, 1998, pp. 123–132. [Online]. Available: http://doi.acm.org/10.1145/285305.285320

[4] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying gpu microarchitecture through microbenchmarking," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2010.

[5] L. Williams, "Pyramidal parametrics," in *Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '83. New York, NY, USA: ACM, 1983, pp. 1–11. [Online]. Available: http://doi.acm.org/10.1145/800059.801126

[6] A. Schilling, G. Knittel, and W. Strasser, "Texram: A smart memory for texturing," *IEEE Comput. Graph. Appl.*, vol. 16, pp. 32–41, May 1996. [Online]. Available: http://dl.acm.org/citation.cfm?id=616080.618958

[7] M. Segal and K. Akeley, *The OpenGL Graphics System: A Specification (Version4.2)*. The Khronos Group, 2012.

[8] K. Fatahalian, S. Boulos, J. Hegarty, K. Akeley, W. R. Mark, H. Moreton, and P. Hanrahan, "Reducing shading on gpus using quad-fragment merging," *ACM Trans. Graph.*, vol. 29, no. 4, pp. 67:1–67:8, Jul. 2010. [Online]. Available: http://doi.acm.org/10.1145/1778765.1778804

[9] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-Time Rendering*, 3rd ed. AK Peters Ltd., 2008.

[10] D. Peachey, “Texture on demand,” Pixar, Tech. Rep., 1990.

[11] M. D. McCool, C. Wales, and K. Moule, “Incremental and hierarchical hilbert order edge equation polygon rasterizatione,” in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, ser. HWWS ’01. New York, NY, USA: ACM, 2001, pp. 65–72. [Online]. Available: http://doi.acm.org/10.1145/383507.383528

[12] C. Wittenbrink, E. Kilgariff, and A. Prabhu, “Fermi GF100 GPU Architecture,” *Micro, IEEE*, vol. 31, no. 2, pp. 50 –59, march-april 2011.

[13] M. Mantor and M. Houston, “AMD Graphic Core Next,” in *High Performance Graphics, Hot3D presentation*, 2011.

[14] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, “NVIDIA Tesla: A Unified Graphics and Computing Architecture,” *Micro, IEEE*, vol. 28, no. 2, pp. 39 –55, march-april 2008.

[15] A. Beers, M. Agrawala, and N. Chadda, “Rendering from Compressed Textures,” in *Proceedings of ACM SIGGRAPH 96*, 1996, pp. 373–378.

[16] G. Knittel, A. G. Schilling, A. Kugler, and W. Straßer, “Hardware for Superior Texture Performance,” *Computers & Graphics,*, vol. 20, no. 4, pp. 475–481, 1996.

[17] J. Torborg and J. Kajiya, “Talisman: Commodity Real-time 3D Graphics for the PC,” in *Proceedings of SIGGRAPH*, 1996, pp. 353–364.

[18] K. Iourcha, K. Nayak, and Z. Hong, “System and Method for Fixed-Rate Block-Based Image Compression with Inferred Pixel Values,” US Patent 5,956,431, 1999.

[19] J. Ström and T. Akenine-Möller, “iPACKMAN: High-Quality, Low-Complexity Texture Compression for Mobile Phones,” in *Graphics Hardware*, 2005, pp. 63–70.

[20] H. Igehy, M. Eldridge, and K. Proudfoot, “Prefetching in a texture cache architecture,” in *Proceedings of the Eurographics/SIGGRAPH Workshop on Graphics Hardware*, 1998.

[21] P. Ganestam and M. Doggett, “Auto-tuning Interactive Ray Tracing using an Analytical GPU Architecture Model,” in *Proceedings of the Fifth Workshop on General Purpose Processing on Graphics Processing Units*, Mar. 2012.

[22] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, “GPU computing,” *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[23] A. Munshi, B. R. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg, *OpenCL Programming Guide*. Addison Wesley, 2011.

[24] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, “Larrabee: a many-core x86 architecture for visual computing,” *ACM Trans. Graph.*, vol. 27, no. 3, pp. 18:1–18:15, Aug. 2008. [Online]. Available: http://doi.acm.org/10.1145/1360612.1360617