2018 IEEE International Symposium on High Performance Computer Architecture

Perception-Oriented 3D Rendering Approximation for Modern Graphics Processors

Chenhao Xie, Xin Fu
ECMOS Lab, ECE Department
University of Houston
cxie, xfu6@uh.edu

Shuaiwen Leon Song
High Performance Computing Group
Pacific Northwest National Lab (PNNL)
Shuaiwen.Song@pnnl.gov

***Abstract*—Anisotropic filtering enabled by modern rasterization-based GPUs provides users with extremely authentic visualization experience, but significantly limits the performance and energy efficiency of 3D rendering process due to its large texture data requirement. To improve 3D rendering efficiency, we build a bridge between anisotropic filtering process and human visual system by analyzing users' perception on image quality. We discover that anisotropic filtering does not impact user perceived image quality on every pixel. This motives us to approximate the anisotropic filtering process for non-perceivable pixels in order to improve the overall 3D rendering performance without damaging user experience. To achieve this goal, we propose a perception-oriented runtime approximation model for 3D rendering by leveraging the inner-relationship between anisotropic and isotropic filtering. We also provide a low-cost texture unit design for enabling this approximation. Extensive evaluation on modern 3D games demonstrates that, under a conservative tuning point, our design achieves a significant average speedup of *17%* for the overall 3D rendering along with *11%* total GPU energy reduction, without visible image quality loss from users' perception. It also reduces the texture filtering latency by an average of *29%*. Additionally, it creates a unique perception-based tuning space for performance-quality tradeoffs on graphics processors.**

***Keywords*-GPU; Approximate Computing; 3D Rendering; User-Oriented Study**

I. Introduction

With the advancement of 3D rendering technology on modern graphics processing units (GPUs), graphical applications in real-time gaming and Virtual Reality are able to deliver photo-realistic scenes and provide extremely authentic user experience by combining 3D geometric model and 2D texture through texture filtering process [1], [2], [3]. As the last stage of the texture filtering execution pipeline, *anisotropic filtering* (AF) accurately depicts the natural material appearance for *any viewing angle*, providing the highest texture quality [4], [5]. It has been widely adopted in 3D rendering to enhance image sharpness and preserve great details for angularly-varying appearance, which is often required for 3D gaming and virtual reality. Commonly, disabling AF or reducing its sampling size can seriously hurt user experience. However, to provide high image quality, AF could drastically degrade 3D rendering's performance and energy efficiency due to heavy computation and excessive texel fetching. Such inefficiency caused by AF is ever significant due to the growing rendering effects and frame resolution.

There have been general studies focusing on improving the efficiency of 3D rendering process through reducing workloads and data movement [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17]. Some of them [8], [9], [10], [12], [13], [17] (❶ in Figure 1) leverage data similarity approximation to enable better reuse and avoid redundancy between image frames (e.g., memorization) without being directly guided by user perception while others [7], [11], [14], [16] (❷ in Figure 1) investigate the relationship between users satisfaction degree and post-rendering output in order to optimal the rendering parameters (e.g., adjusting resolution) to trade off image quality with performance . Meanwhile, recent works [12], [18], [19], [20], [21] in the field of computer architecture have suggested that texture fetching significantly bottlenecks 3D rendering process. However, these previous studies ignored an important correlation which is that *streams of bits in architecture are closely related to user perception*, as shown as ❸ in Figure 1. How to correlate data approximation with human visual system in 3D rendering is still unclear, leaving a knowledge gap.

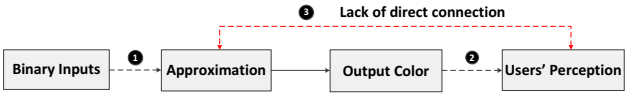

Figure 1: State-of-the-art rendering methods ignore the direct connection between user perception and data approximation.

To bridge this gap, we explored AF's impact on users' perception of image quality in a quantitative manner through the Structure Similarity [22] index (SSIM), a state-of-the-art metric for evaluating perceived quality of a image. By comparing the gaming images with AF enabled and disabled, we found that SSIM index map can generally reflect user perception difference. We also observed that AF only affects user perception on a portion of pixels in an image frame. This observation motives us to propose an approximation method that only enabling AF for the perceivable pixels to reduce unnecessary texel fetching and computation.

To achieve this targeted filtering, we evaluated the available design choices and decided to use hardware-based approximation strategy due to its low design and runtime cost, better control granularity, and capability of leveraging fine-grained low-level details (e.g., texel attributes). Another challenge we face is that conventional SSIM index map is generated after going through AF so we cannot apply it for runtime prediction. To overcome this, we thoroughly investigated how GPU hardware samples for different filtering methods and explored the root cause for pixel-level difference before and after AF. Based on these insights, we propose a runtime SSIM index prediction method named **AF-SSIM** with a reconstructed AF-sensitive SSIM formula. We also propose two runtime prediction schemes to support AF-SSIM: *sample-area based prediction* and *distribution based prediction*. Based on these two prediction schemes, we propose the **P**erception-**A**ware **T**exture **U**nit (**PATU**) design and its optimizations, which maximally reduces unnecessary AF workloads while maintaining high image quality. To the best of our knowledge, this is the first work that proposes

2378-203X/18/$31.00 ©2018 IEEE
DOI 10.1109/HPCA.2018.00039

architecture-level design that enables efficient runtime AF approximation for improving the overall 3D rendering efficiency on a native GPU without scarifying user experience. This design also creates a unique tuning space for performance-quality tradeoff directly guided by user perception, which would not otherwise exists. Finally, we designed a simple user study to evaluate the impact of PATU on users' experience through trace-based game replay. This attempt aims to further extend the prior PATU evaluation which is based on a more general scientific metric (i.e., SSIM) to include more inputs from users' perception. Specifically, this paper makes the following contributions:

- We conducted thorough analysis on AF's impact on the 3D rendering efficiency and user perception. We found that AF only impacts user perception on some pixels in a frame, hinting approximation opportunities.
- After a thorough analysis on hardware sampling method for the filtering process on GPU, we discovered two important filtering features that can help predict the similarity degree of two pixels and determine if AF can be approximated at runtime.
- We proposed a cost-effective perception-aware texture unit design to approximate texture filtering process at hardware level. We further optimize the design to improve image quality through exploring data locality in the texture cache.
- We evaluated our proposed design by rendering seven real-world 3D games with different resolutions. We provided detailed design space exploration, design impact analysis and user experience study. Under a conservative threshold, our design achieves a significant average speedup of **17%** (up to **24%**) for 3D rendering along with **11%** (up to **16%**) total GPU energy reduction (i.e., DRAM is included), and an average of **29%** (up to **42%**) texture filtering latency reduction, without visible quality loss (MSSIM $\geq$ **93%**). It achieves even better performance and energy efficiency for games with higher resolutions.

II. Background And Motivation

A. Modern 3D Rendering Pipeline

Traditionally, GPUs were designed as the special-purpose graphics processors for modern 3D rendering process, which is the computer graphics process using 3D vertex data to create 2D images with 3D effects. Figure 2 illustrates a general 3D rendering architecture for contemporary rasterization-based GPUs, which serves as the baseline for this work. It employs unified shader model [23] which consists of programmable core clusters for vertex and fragment processing for better GPU utilization. Each cluster corresponds to a single texture unit.

During the rendering process, input vertexes are first fetched from memory into the vertexes cache. These vertexes are processed during *Vertex Processing* in Unified Shader to calculate their attributes including position, color and texture coordinate. These vertexes are further transformed and assembled into triangles. They pass through geometry-related kernels such as *Clipping, Face Culling and Tessellation* to generate extra triangles or remove non-visible triangles. After this, triangles are sorted into tiles base on their positions in *Titling Engine*. Each tile contains a small number of triangles so that all the pixels of a tile can be stored in local on-chip

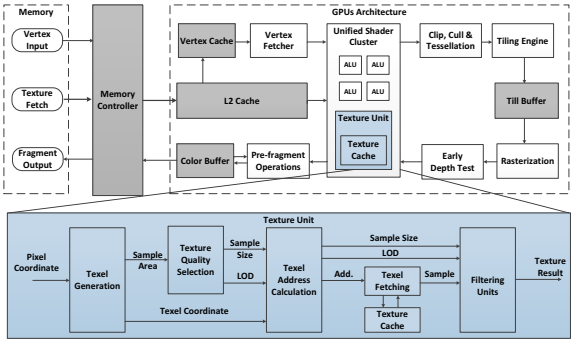

Figure 2: A general 3D rendering architecture for modern rasterization GPUs, with a zoom-in view on the texture unit.

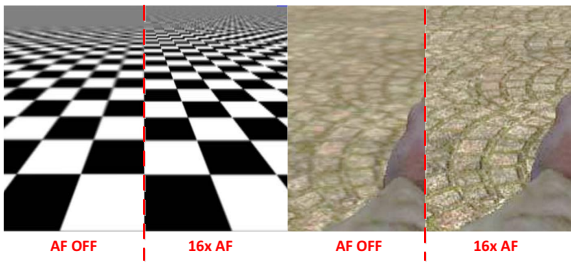

Figure 3: Improved image quality through anisotropic filtering.

memory. These tiles are then scheduled as basic execution units and fed into *Rasterization Process* to generate fragments, each of which is equivalent to a pixel in a 2D image. These fragment tiles then go though the *Early Depth Test* to remove overlapped fragments to reduce memory access.

The final fragment attributes such as color and texture are computed through *Fragment process* in Unified shader. During this phase, the shaders fetch extra information by sampling texture data from the texture units (the highlighted blue block) for better image fidelity. When all the fragments in the same tile are rendered, their corresponding pixel values are sent to the fragment buffer once per tile to enhance responsiveness and reduce memory transfer [19].

B. Impact of AF on Modern 3D Rendering

Texture filtering within the fragment processing (the blue block in Figure 2) is one of the most memory-intensive phases in 3D rendering [12], [19], [20], [21]. It calculates the weighted averaged value of sampled texels that best approximate the correct texture color of a pixel [23], [24], [25]. Under user-defined texture filtering method, the *Texel Generator* first calculates texel coordinates and the sample size of texels using pixel coordinates. Then the sample size is passed through the *Texture Quality Selector* to define the level of details (LOD) for the texels. Under the selected LOD, *Texture Unit* will sample texels from the pre-defined texture map. After that, memory address for each required texel (i.e., pixel of the texture) will be calculated by *Texel Address Calculator* using texels' attributes. Based on this information, *Texel Fetching Units* fetch the actual texels. If cache hits, texture unit reads the texels from the texture cache. Upon misses, it fetches the texels directly from the off-chip memory. Finally, when all the texels of the requested texture are collected, its four-component (RGBA) color is calculated and the texture unit returns to the shaders with the filtered texture results.

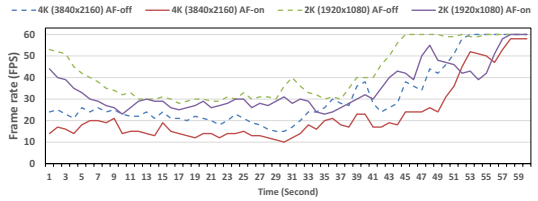

Figure 4: Impact of AF when running R.Bench on iPhone 7s.

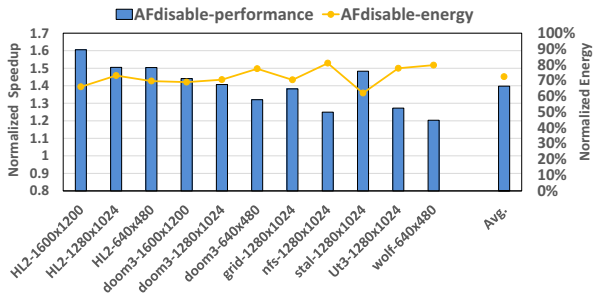

Figure 5: Normalized speedup and energy reduction of 3D rendering when AF is disabled.

To improve image quality of gaming and virtual experience (e.g., reducing blurriness [4], [5]), contemporary GPUs add a high quality filtering method called *Anisotropic Filter* (AF) [21], [26] as the last step of texture filtering process, which joins right after bilinear and trilinear filtering [24] to form a three-step texture-filtering process. AF can effectively enhance the sharpness of the textures on the surface that are at oblique viewing angles with respect to the camera [27]. For example, Figure 3 demonstrates the visual difference between enabling and disabling AF. At extreme viewing angles when pixels on the screen extend far away from the viewer or camera, taking equal number of samples on each axis from the texture map would result in texture blurriness along the axis. Due to its aliasing elimination effect, AF has become one of the most essential filtering techniques providing high-quality images for PC/mobile gaming [5] and Virtual Reality [3].

Despite AF's great advantages, it may incur significant amount of computation and memory accesses, thus degrading the overall performance and energy efficiency of 3D rendering. This is because AF often requires a large area of texels to filter a particular pixel with certain camera angle, and the number of required texels greatly increases with the size of the sampling area [6]. To limit the sampling size, the maximum support level of AF is often defined for the texture unit, e.g., the max level of AF implemented in today's texture unit [26] permits 128 texels to filtering the color for one pixel, which is **16X** of the texels required by trilinear filtering. To further quantify AF's performance and energy impact on 3D rendering, we conducted a series of evaluation using real-world games on both mobile platform and popular cycle-accurate simulator, i.e., (1) recording the frame rate (fps) of the *Relative Benchmark* (R.Bench) [28] on iPhone 7 plus which integrates the last A10 GPUs (PowerVR Rogue Architecture [29]); and (2) executing several 3D games on a graphics simulation platform with AF enabled/disabled (refer to Section VI for the detailed experimental setup). R.Bench is a next-gen rendering benchmark demanding higher texture quantity than that on today's mobile platforms. It is fully configurable using OpenGL_ES3 extension. We summarize the following observations:

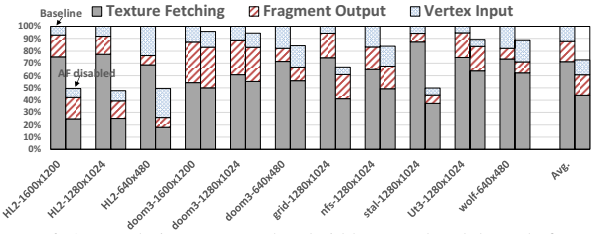

Figure 6: 3D rendering memory bandwidth usage breakdown before and after disabling AF. Left is the baseline with AF on while right is with AF off.

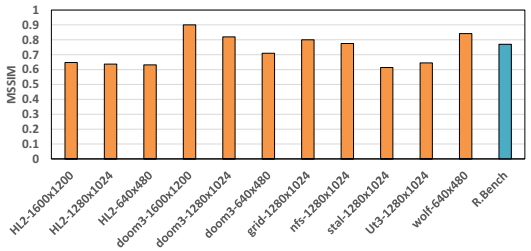

Figure 7: Impact of AF's absence on user perception.

(1) Overall Impact on Performance and Energy Consumption. Figure 4 shows the fps of R.Bench with 2K and 4K resolution. We tested it multiple times with AF on and off. The results demonstrate that R.Bench can not satisfy the real-time performance requirement (60 fps) for most of its frames because the benchmark leverages high-quality color effects and requires a large size of texture data. Disabling AF can significantly improves the frame rate by **21%** (up to 54%) for 2K and **43%** (up to 83%) for 4K. Furthermore, Figure 5 shows the normalized speedup and energy reduction of 3D rendering on our simulation platform when AF is disabled. These results show a similar performance impact as what is demonstrated by R.Bench on iPhone 7s. The performance of 3D rendering is improved by **41%** on average (up to 60%) with an average of **28%** energy reduction (up to 33%). In summary, both Figure 4 and 5 prove that AF significantly limits the 3D rendering's performance and energy efficiency for high-end mobile platforms, and such impact is exacerbated for higher resolution scenarios.

(2) Root Causes of Negative Performance and Energy Impact. We observe that AF poses not only high computation workloads (i.e., large sampling size of texels) but also significant memory bandwidth requirement during texture filtering process. Figure 6 shows the memory access breakdown for 3D rendering. Since resolutions may affect memory access patterns, we also tested different resolutions for the selected games. The figure demonstrates that the texture fetching process in 3D rendering accounts for an average of **71%** of the total memory bandwidth usage. Disabling AF significantly reduces the memory access, particularly through texture fetching (i.e., an average reduction of **28%** and up to 51%), suggesting a drastic reduction in sampling size of texels for filtering. This also results in texture filtering latency reduction by an average of **47%**.

C. Perception Impact of AF

Before proposing any AF optimization, we need to understand its impact on users' perception in a quantitative manner so its performance-quality tradeoffs on GPU can be investigated. If a tuning space for such tradeoffs exists, we may systematically approximate AF for less computation and

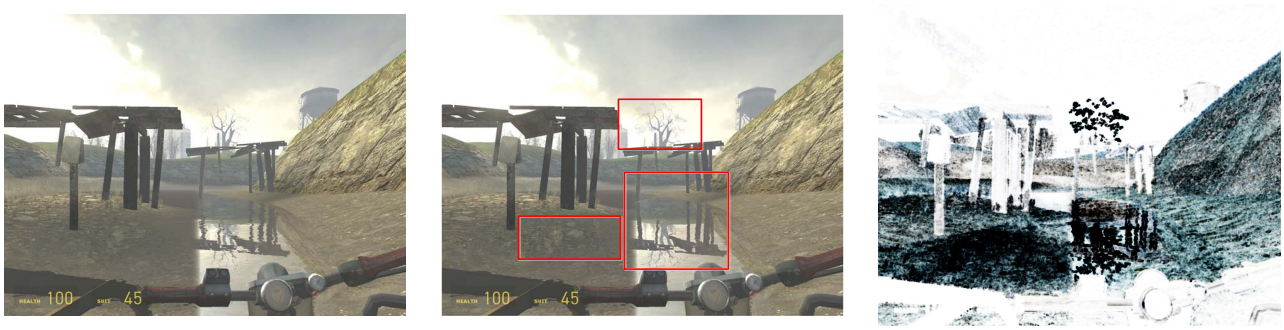

Figure 8: Perception impact of AF-on/-off on a selected frame from Half-Life 2 with resolution of 1600x1200: (left): AF-on; (middle): AF-off; (right): their SSIM index map (lighter areas indicate higher SSIM values for pixels and high similarity before and after AF).

memory fetching. Thus, we incorporate Structure Similarity (SSIM) [22] index, a state-of-the-art metric for evaluating *perceived quality of digital images and videos*, as our user-perception measurement method. SSIM provides a measure of visual closeness between an image and its reconstructed version. Since it combines three independent components in human visual system (HVS) [30], [31] including *luminance, contrast and structure comparison*, SSIM has been widely adopted in evaluating user perceived quality of image and outperforms other assessment indexes in this aspect (e.g., peak signal-to-noise ratio and mean squared error). For our evaluation, we denote Y as the pixel value under 16x AF and X as the pixel value with AF disabled. The SSIM between Y and X can then be formulated as Eq. (1):

$$\begin{aligned} SSIM(X,Y) &= \frac{(2\mu_x\mu_y)}{(\mu_x{}^2+\mu_y{}^2)}\frac{(2\sigma_x\sigma_y)}{(\sigma_x{}^2+\sigma_y{}^2)}\frac{(\sigma_{xy})}{(\sigma_x\sigma_y)} \\ &= \frac{(2\mu_x\mu_y+C_1)(2\sigma_{xy}+C_2)}{(\mu_x{}^2+\mu_y{}^2+C_1)(\sigma_x{}^2+\sigma_y{}^2+C_2)} \end{aligned} \tag{1}$$

where $0 \leq SSIM(X,Y) \leq 1$, when $X,Y \geq 0$, μ_x and μ_y denote the means of X and Y, σ_x and σ_y denote the standard deviation of X and Y, and σ_{xy} denotes the cross-correlation between X and Y. To avoid division by zero, positive constants C_1 and C_2 are added for the purpose of stability. Since an image usually consists of millions of pixels, the mean SSIM index (*MSSIM*) is commonly used to evaluate the overall image quality:

$$MSSIM(X,Y) = \frac{1}{M}\sum_{j=1}^{M} SSIM(x_j,y_j) \tag{2}$$

Figure 7 illustrates the impact of disabling AF on users' perceived quality of gaming images through MSSIM index. It shows that disabling AF can pose drastic damage to users' perceived image quality by **28%** on average (up to 39%). Naively sacrificing image quality and user experience to trade performance/energy efficiency is often unacceptable.

D. Approximation Hints through SSIM Map

Since directly disabling AF is not a viable solution, we resort to the pixel-level SSIM map for understanding user perceived image quality. Figure 8 shows an example frame from Half-Life2 (1600x1200 resolution) when AF is enabled (*left*), disabled (*mid*), and their SSIM index map (*right*). In the SSIM index map, the lighter color represents the higher reconstructed image area with larger SSIM values which indicates less image quality difference after AF. On the contrary, the darker the color is, the more different a image is perceived before and after AF. Comparing the three subfigures, we find that SSIM index map can successfully preserve the perceived image quality change when disabling AF (red-box areas), e.g., disappearance of the mirrored water rippling effects, blurriness of the lawn and remote mountain texture, and loss of the smoking effects that covers portions of leaves on the tree far away. Furthermore, we observe that *disabling AF only affects user perception on a portion of the image*. Specifically, more than half of the pixels (lighter areas) in this SSIM index map still exhibit high perceived quality without AF. This observation motivates us to optimize 3D rendering's efficiency by approximating AF's filtering process for the *non-perceivable pixels* to significantly reduce AF's workloads and the overall texel inputs for texture filtering.

III. Design Choice

Motivated in Section II-D, our basic design goal is to approximate AF for *non-perceivable pixels* in a image frame. We evaluate two potential design choices: software-based and hardware-based approximation. Specifically for reaching our goal of approximating AF to enable efficient runtime selective filtering, we choose hardware-based approximation due to the following reasons:

(I) Runtime Cost. Hardware-implemented AF has much lower runtime cost than its software counterpart. For instance, prior work [6] has pointed out that when implementing a commonly-applied filtering algorithm, hardware-based AF significantly outperforms the software-based AF (e.g., 1.5X throughput). Thus, software-based AF is not suitable for real-time 3D rendering, especially in user-interactive gaming and VR. This is why modern commercial GPU hardware overwhelmingly adopts hardware-based AF [13], [23], [29].

(II) Control Granularity. Programmers usually cannot directly control or modify the texture filtering functions from graphics libraries that are implemented in hardware texture unit (Figure 2). For instance, developers can only call graphics APIs such as `GL_EXT_texture_filter_anisotropic()` in OpenGL [32] to perform AF. Such software modularization does enable less-complex high-level programming but also significantly limits optimization opportunities that may be achievable through fine-grained control.

(III) Targeted Filtering. Software approximation can not exactly capture the runtime texture attributes which are used to enable targeted filtering. As we discussed in Section II-D, disabling AF on different texture-shaded pixels may have quite different influence on user perception. However, software methods can not acquire fine-grained runtime data

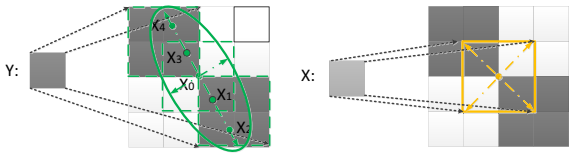

Figure 9: Inverse mapping of pixel Y (enabling AF) and X (disabling AF) from screen to the texture map. The example texture map comprises of 16 (4×4) texels.

such as texel coordinates and fetching addresses to effectively distinguish the texture information since all these data are hardware-level intermediate results from runtime execution. Due to this reason, software methods [6] have to treat all the textures equally, which is obviously against our key idea of only processing user-perceivable pixels.

IV. Perception-Oriented Approximation

As shown in Figure 8, it would be highly preferred to have such a computed SSIM index map at runtime to assist targeted AF filtering on only perceivable pixels. However, the original SSIM method cannot be directly applied at runtime to measure the similarity of pixels without rendering the entire image. In other words, we cannot use the rendering result with AF to predict the already completed rendering process. In this section, we propose a runtime SSIM index approximation method named **AF-SSIM** using several special features from the texture filtering process on modern GPU hardware.

A. Hardware-Level Filtering Method

As we discussed in Figure 3, the SSIM degradation after disabling AF is due to the blurriness when GPU renders anisotropic objects using isotropic filtered texture (e.g., trilinear filtering). But Figure 8 also demonstrates that there are a large portion of pixels do not actually require AF to achieve perceivable quality. In other words, isotropic filterings (e.g., bilinear and/or trilinear) prior to AF is already adequate for these pixels. Thus, the fundamental goal of AF-SSIM is to runtime predict whether a pixel needs AF or not without the need of going through AF to generate SSIM index map. If we assume that X is the color of a rendering pixel with only trilinear filtering (TF for short) while Y is the texture color when X is further filtered by AF, AF-SSIM is to runtime identify how different X and Y are in terms of their texture color, which requires the knowledge on how pixels are actually filtered in GPU hardware.

Figure 9 illustrates the hardware filtering methods for calculating a pixel's texture color under TF (pixel X) and AF (pixel Y). The two small squares on the left represent the texture colors of pixel X and Y while the two large squares on the right represent the two pixels' corresponding texture sampling space which consists of 16 texels (i.e., a 4×4 texture map). During texture filtering, texture units (Figure 2) normally leverage mipmapping to sample necessary texels from a two-tiered texture map. For simple illustration, we show the texture mapping on only one texture map in Figure 9. As shown in the figure, for TF, GPU texture units first calculate the length of the two vertical diagonal lines of a sampling area based on pixel X's coordinates and then maps this square shape area decided by the two diagonal lines onto the texture map to determine which texels should be used to filter X. Then, these texels will be sampled from the two-level texture map and their weighted average value will be X's texture color. On the contrary, for AF, a texture unit samples the texels from an eccentric ellipse shape. It first calculates the lengths of the major and minor axes in the ellipse, and then determines AF's sample size N using the ratio of the major axis to the minor axis. Based on the sample size, it generates multiple trilinear samples (X_i) along the major axis. The value of X_i is calculated using the exactly same TF method to compute pixel X (i.e., AF uses multiple TF's outputs as inputs). For example, the three green dash squares in the figure are the sample areas of X_i . Finally, Y's texture color is calculated by averaging the values of multiple trilinear samples (X_i), as shown in Eq.(3):

$$Y = \frac{1}{N}\sum_{i=0}^{N-1} X_i = \frac{1}{N}\sum_{i=0}^{N-1}\sum_{j=0}^{7} \omega_{ij} t_{ij} \tag{3}$$

Where t_{ij} and ω_{ij} are the texel colors and texel weights used to filter the value of X_i, and N is the sample size determined by the axis ratio of anisotropic. Note that Y and X share the same center coordinate in our example which is X_0. Also, since X_i and X share some texels from the same texture map, they exhibit certain value similarity (Section IV-C(B)).

B. Constructing AF-SSIM Formula

Using the same example in Figure 9, we define ∇_i as the ratio of X_i to X. Thus the texture color of pixel Y (with AF) is calculated as a function of X (without AF) and μ_∇, shown in Eq.(4):

$$Y(X,\mu_\nabla) = \frac{1}{N}\sum_{i=0}^{N-1} X_i = \frac{1}{N}\sum_{i=0}^{N-1}(\nabla_i X) = X * \frac{1}{N}\sum_{k=0}^{N-1}\nabla_i = \mu_\nabla X \tag{4}$$

From Eq.(4), we can immediately observe that if $X \neq 0$, the ratio of Y to X is equal to μ_∇ which is the mean of ∇_i. We denote μ_∇ as the **similarity degree** between Y and X. Combine this correlation with the original SSIM formula Eq.(1) in Section II-C, we define the new AF-based Structure Similarity index (AF-SSIM) as a function of the similarity degree.

$$AF\_SSIM(X,Y) = AF\_SSIM(\mu_\nabla) = \left(\frac{(2\mu_\nabla + C_1)}{(\mu_\nabla{}^2 + 1 + C_1)}\right)^2 \tag{5}$$

Similar to the classic SSIM, Eq.(5) represents the perceived image quality difference of a pixel between enabling and disabling AF. Although the formula is significantly simplified to only rely on the similarity degree μ_∇ ($C1$ is a positive constant), it still cannot be directly used for runtime SSIM index prediction to avoid AF because obtaining the exact value of μ_∇ requires going through AF. Thus, we need to find an alternative parameter to approximate μ_∇ which has to satisfy the following criteria: (a) it can represent the strong correlation between X and Y; (b) it should not be correlated with texture values so it can be acquired before the actual texel fetching and filtering; (c) it should be calculated using the existing texture attributes to minimize the hardware overhead.

C. AF-SSIM Runtime Prediction Methods

(A) Sample-Area Based Prediction (AF-SSIM(N)). Based on the filtering methods discussed in Section IV-A, we find that a simple way to approximative the similarity degree μ_∇ is to calculate the sample area difference between TF and AF. As discussed previously, the sample area of AF

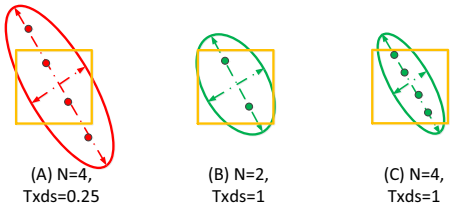

Figure 10: Two prediction schemes: sample-area based prediction ((A)+(B)) and distribution-based prediction ((A)+(C)). N is sample area difference and $Txds$ is the texel distribution difference.

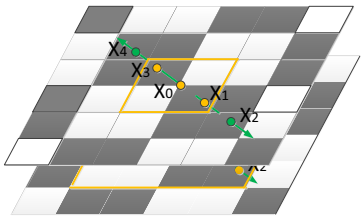

Figure 11: A concentration of AF's input samples appears within TF's sample area.

is an ellipse which is determined by its major and minor axes while the sample area of TF is a square with equivalent diagonal lines. In this prediction scheme, we simply calculate the ratio of the major axis to the minor axis of the ellipse, which has been defined as the sample size N, to predict μ_∇. As shown in Figure 10(A) and (B), the smaller sample size N is (i.e., N is closer to 1), the sample area of AF is closer to a square (i.e., the more the ellipse grows closer to a circle) which suggests the final texture color filtered by AF is very similar to the texture color through TF ($Y \approx X$). More importantly, sample size N for AF can be directly obtained right after *Texel Generation* in the filtering pipeline (Figure 2), which is way prior to the actual texel fetching and filtering. This significantly increases runtime efficiency (more discussion in Section V). Therefore, we can apply N to represent the sample area difference and use it to replace μ_∇ in Eq.(5). As a result, the sample-area based prediction AF-SSIM(N) can be denoted as

$$AF\_SSIM(N) = (\frac{2N}{N^2+1})^2, \qquad where \quad 1 \le N \le 16 \tag{6}$$

(B) Distribution Based Prediction (AF-SSIM(Txds)). Although sample-area based prediction can be an effective way to indicate the filtering difference between TF and AF by distinguishing the pixels that have narrow texture sample area, it does not consider the **relative similarity** of texels used for TF and AF within the sample area. For instance, Figure 10(C)) shows the case that it has the same sample size N as case (A) but most of its sampled texels are shared within TF's sample area (yellow square). Pixels with this special feature also do not need to go through AF since the similarity degree μ_∇ betwen X and Y is very high. Figure 11 further illustrates this scenario where majority of the input samples (e.g., X_0, X_1 and X_3) required for AF are located in the same sample area of TF. Because of the distribution concentration, the 8 texels from the two yellow squares are the major factors determining the color of the texture. Eq.(7) demonstrates that the more X_i are concentrated in TF's sample area, the close Y and X gets. Since X_1 and X_3 use the same texels with X_0, we can approximate the value of X_1 and X_3 using X_0. In this case, the difference between Y and X is related to

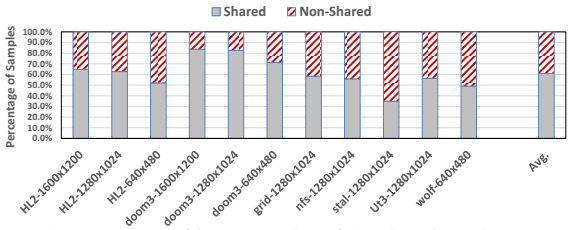

Figure 12: Percentage of input samples of AF that share the same set of texels with TF during 3D rendering.

how many trilinear samples do not share the same texels. Furthermore, to evaluate how often such relative similarity among texels occurs, we explore its probability in real-world games. Figure 12 shows that **an average 62%** of AF's input samples share the same set of texels with TF during 3D rendering. Therefore, considering texel similarity in addition to sample area can further reduce the pixels going into AF.

$$Y = \frac{1}{5}\sum_{i=0}^{4} X_i \approx X + \frac{1}{5}\sum_{j=2,4}(\nabla_j - 1)X \tag{7}$$

To measure how concentrated AF's samples locate inside of TF's sampling area at runtime, we leverage the concept of *entropy* in math to describe the AF's texel distribution. Entropy is often used to describe the accuracy of probability. Eq.(8) shows its math form, where P_i denotes the probability of event i and M denotes the number of events. Its lower boundary is 0 when probability P_i is equal to 1 (i.e., the event definitely happens), and its upper boundary is $log_2(M)$ when the event has a unified distribution (i.e., no specific event is more likely to occur).

$$Entropy(P) = -\sum_{i=0}^{M-1} P_i log_2(P_i) \tag{8}$$

We apply the entropy of texel distribution to represent the accuracy of using a subset of texels to replace the entire set of texels in the anisotropic sample area. In this definition, P_i is defined as the probability of the TF's texels that are shared by the samples of AF. For example, in Figure 11, three AF's samples share the texels in TF's sample area and the other two samples use different set of texels. For this pixel, the *probability vector* of sample texels (or events) is $\{0.6, 0.2, 0.2\}$. The number of vector elements in this vector depend on the number of TF's sample areas that AF's samples overlap with. We then can calculate the entropy of texel distribution, $Entropy_{tx}$, using the probability vector.

Since our goal is to approximate the difference between pixel X and Y (μ_∇) in order to runtime predict AF-SSIM, we define a metric named **texel distribution similarity** ($Txds$) to describe the texel distribution difference between AF and TF, shown in Eq.(9). Note that we normalize the entropy to its upper boundary based on the sample size N so that we have the same measurement interval for different sample sizes. When $Entropy_{tx}$ is close to 0, texel similarity between AF and TF is high ($Txds$ is close to 1) which indicates that AF is not necessary. Otherwise, when $Txds$ is close to 0, texel similarity between AF and TF is low which suggests AF is needed to maintain quality.

$$Txds(P,N) = 1 - Entropy_{tx}(P)/log_2(N), \quad 0 \le Txds(P,N) \le 1 \tag{9}$$

Finally, we use Txds to predict μ_∇ and calculate AF-SSIM as

$$AF\_SSIM(Txds) = (\frac{2Txds(P,N)}{Txds(P,N)^2+1})^2 \tag{10}$$

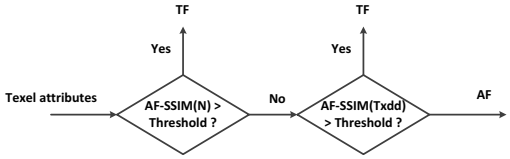

Figure 13: Overall prediction flow.

Note that the runtime information for predicting Eq.(10) is P_i and N. As discussed previously, N can be obtained right after *Texel Generation* stage (Figure 2). Similarly, P_i can be calculated after *Texel Address Calculation*. Both of them can be predicted prior to the actual filtering process.

(C) Put It All Together (Prediction Flow). Figure 13 shows the overall flow for enabling runtime structure similarity prediction to avoid unnecessary AF. The filtering process will go through two threshold-checking stages, reflecting the two AF-SSIM runtime prediction methods. One occurs after Texel Generation using $AF\_SSIM(N)$, and the other occurs after Texel Address Calculation using $AF\_SSIM(Txds)$. If the predicted AF_SSIM is greater than the thresholds presented in these two stages, we mark the pixel as approximated and only conduct TF. Otherwise, we consider AF is necessary for image quality and proceed with AF as the final texture filter. To simplify the design, we use a unified threshold for both $AF\_SSIM(N)$ and $AF\_SSIM(Txds)$ because both methods share the same objective which is to approximate similarity degree μ_∇. This also significantly reduces a large complex tuning space. The discussion on threshold tuning will be shown in Section VII-A.

V. Perception-Aware Texture Unit Design for AF Approximation

In this section, we first propose the Perception-Aware Texture Unit (**PATU**) design based on the two runtime AF-SSIM prediction schemes, and then discuss its optimizations.

A. Structure of PATU

Figure 14 presents the architecture diagram of PATU. The perception-aware functionality of PATU is implemented on top of the conventional texture unit shown in Figure 2. It consists of three new components: ❶ sample area similarity checking to distinguish the pixels with small sample size N; ❷ a hash table to runtime gather texel address information and count the number of shared texels; and ❸ approximation control for address entropy which computes texel distribution similarity $Txds$.

Sample area similarity checking ❶ is a simple compute logic. It receives the sample area information N from *Texel Generation* and applies it in Eq.(6) to predict $AF\_SSIM(N)$. It then compares $AF\_SSIM(N)$ with a pre-defined threshold (either user-defined or the optimal from design space exploration) to decide whether AF is necessary for this pixel.

The runtime texel address hash table ❷ is an address deferred buffer (acting like a cache) responsible for evaluating AF's texel distribution P_i in Eq.(8). It is implemented as a fully associate SRAM cache which concludes 16 entries (i.e., the max AF level is 16 on modern GPUs) and each entry is paired with a count tag. It counts how many times AF's samples share the same set of texels with TF based on the texel addresses of trilinear samples as described in Section

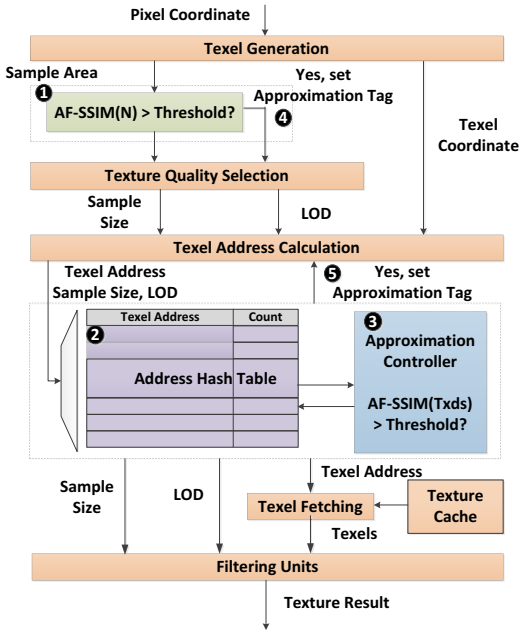

Figure 14: Perception-Aware Texture Unit Design.

IV-C(B). The table is indexed by the trilinear samples' texel addresses and a counter is attached to each entry. Specifically, an incoming input addresses will be compared with the addresses stored in the table from top to bottom. If there is a hit, the count tag of the corresponding entry will be increased by 1. Otherwise, this input address will be stored in the first available entry of the table.

The approximation control for address entropy ❸ is a compute logic that receives the texel address counts from the hash table and the sample size N from *Texel Address Generator*. Using the counts number, it calculates the probability vector and the $AF\_SSIM(Txds)$ of the pixel based on Eq.(10). Then it compares $AF\_SSIM(Txds)$ with the threshold to determine whether the pixel can be approximated.

B. Detailed Filtering Process of PATU

In modern rasterization-based GPUs, four pixels are grouped as a quad and processed under the SIMD model within the texture unit. Each quad is considered as one basic processing unit passing through the entire texture filtering pipeline. Each pixel within the quad has its own independent registers to hold the texel attributes and final result. Recall the AF_SSIM prediction flow in Figure 13, the filtering process will go through two threshold checking stages. As shown in Figure 14, we insert the first checking stage after Texel Generation and the second one after Texel Address Calculation. We propose such design because we expect the prediction for AF_SSIM to finish before the actual texel fetching and filtering. Thus, this efficient design not only avoids the unnecessary AF on non-perceivable pixels but also decreases the texel inputs for the overall filtering.

After finishing the texel generation, the sample size N will be fed into the sample area similarity checking ❶ of PATU. If the predicted AF_SSIM is greater than the threshold, the approximate tag for the pixel is set to 1 indicating that the pixel will be filtered using TF. After that, Texture Quality Selection will determine the LOD of the texture based on

the sample area. Meanwhile, the sample size of pixels which are marked as *non-perceivable* (i.e., no AF) will be set to 1. Then, the sample size and LOD are sent into *Texel Address Generator* to calculate texels' addresses. In this stage, the address ALU computes the texels' addresses for one trilinear sample per loop. After the required texel addresses for one trilinear sample is calculated, PATU feeds them (i.e., total 8x32 bits) into the hash table ❷ immediately to generate the probability vector P_i. To reduce overhead, we overlap the look-up table access with the address calculation.

After all the addresses are calculated and fed into the hash table, the approximation controller ❸ receives P_i from the hash table to calculate $AF\_SSIM(Txds)$ of the pixel and determines whether AF is necessary. If the $AF\_SSIM(Txds)$ is greater the threshold, the controller sends an approximate tag ❺ to *Texel Address Calculation* to recalculate the addresses using sample size $N = 1$ which is the sample size of TF (i.e., readjusting input sample size for TF under the assumption of no AF). If AF cannot be approximated for this pixel, the texel attributes such as sample size N, levels of details (LOD) and texel addresses will be sent to the *Texel Fetching* stage. Meanwhile, it resets the texel address hash table for the next pixel processing request. Note that if the sample size of the processed pixel is 1 (i.e., TF only) or the processed pixel is marked with "approximation tag", the texel attributes will bypass ❷ and ❸ to avoid deadlock. Finally, the *Texel Fetching Unit* gathers all the required texels and send them into the *Filtering Unit* for the actual texture filtering and producing final texture result.

C. Discussion

(1) Divergence within a quad. As discussed previously, four pixels are grouped into a quad and being processed together. Meanwhile, our proposed runtime *AF_SSIM* prediction focuses on individual pixel in order to accurately reflect the actual SSIM index map. Thus, it is possible that pixels in a quad are filtered differently. However, this scenario rarely occurs under our design because the pixels within a quad are often close to each other in terms of pixel coordination, and they usually have the same sample size and level of details. Across all the games in Table II, only an average of **1%** (up to **1.6%**) of the quads exhibit such prediction divergence among their pixels. Even if such divergence occurs, our scheme can still accelerate texture filtering through less texel fetching and reduced memory access latency. Hence, it is unnecessary to design a complex architecture specifically for improving prediction divergence in a quad.

(2) Texture quality shift within a frame. Occasionally, pixels from the same 3D-rendering object may have different AF-SSIMs by prediction. For instance, Figure 15 shows two facades of a building (A and B) rendered in different directions. Facade A faces toward the view camera so that it has relative small sample size and does not require AF under our prediction threshold (e.g., $Threshold = 0.2$ for AF-SSIM) while B is a slant facade which is filtered using AF to reduce blurriness. Level of details (LOD) or texture quality is an important attribute of texel and it is selected based on what filter is applied. Shown in Figure 14, *Texel Quality Selection* outputs a pixel's LOD. The LOD of a pixel is determined by the minor axis of the sample area for AF and the diagonal line of the square area for TF, and AF's minor axis is normally shorter than the diagonal line of TF when they have the same

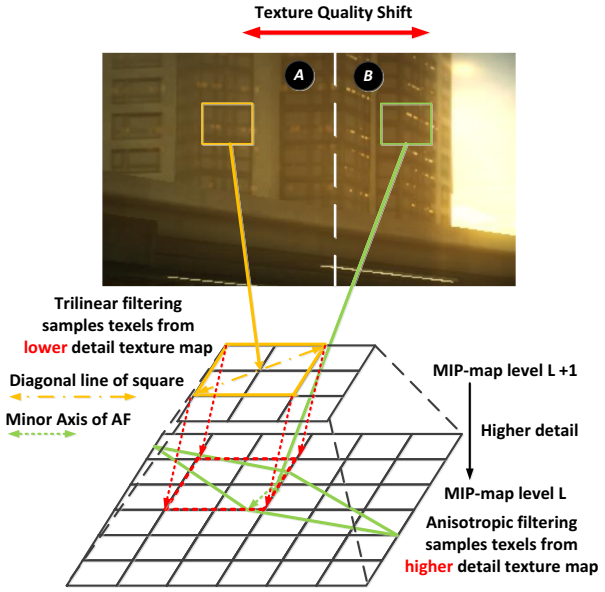

Figure 15: Texture quality (LOD) shifts when AF and TF sample texels from different detail texture maps.

distance to the view camera. As a result, the texels to render facade B in Figure 15 are sampled from the higher detail texture map (i.e., lower MIP-map) than facade A. Therefore, directly using TF to approximate AF causes a clear texture quality shift at the corner of facade A and B (white dashed line). For example, facade A loses some level of details such as lights in the rooms and window textures. To address this LOD shift problem in a frame, PATU reuses AF's LOD (higher texture quality) for the non-perceivable pixels with the approximation tag (i.e., pixels filtered by only TF) by moving TF's sampling level to the same level as AF's without increasing sample size (e.g., lower the yellow square to the same surface of the green ellipse). In this way, PATU not only avoids LOD shift but also leverages data locality in the texture cache through reuse rather than fetching texels from different MIP-maps.

D. Design Overhead Analysis

For area overhead, we apply McPAT [33] to model PATU under 28 nm technology. We observe that PATU only incurs very small hardware overhead compared to the entire GPU area. The major area overhead comes from the added storage, which is the texel address hash table. For each texture unit, there are four texture filtering pipelines working simultaneously for the four pixels in a quad. To store texels' addresses in the hash table, we add four 16-entries look-up table and each entry holds eight 32bit texel addresses, with $(8\times32)+4 = 260$ bits per entry (4 bits are for the count tag per entry). It is approximately 2KB in total per texture unit. Based on McPAT, it occupies about 0.15 mm^2 per unified shader cluster, which is 0.2% of a total 66 mm^2 GPU area. Other added compute logic for calculating AF-SSIM metrics and comparing with threshold only incurs negligible area overhead compared to the hash table. For latency, we use CACTI [34] to model the hash table as a SRAM cache and under our configuration its access latency is estimated as less than 1 cycle. We also estimate the compute latency of the added logic based on the address calculation latency for

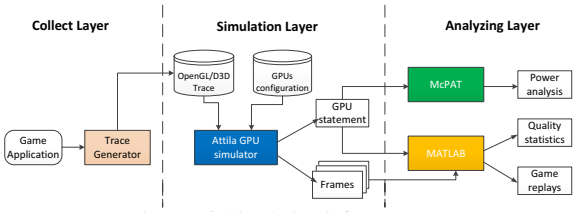

Figure 16: Simulation infrastructure.

TABLE I: BASELINE SIMULATOR CONFIGURATION

Frequency	1GHz
Number of cluster	4
Unified shader per cluster	16
Unified shader configuration	SIMD4-scale ALUs 4 shader elements 16x16 tile size
Number of Texture Units	1 per cluster
Texture unit configuration	4 address ALUs 8 filtering ALUs
Texture throughput	2 cycle per trilinear
Texture L1 cache	16KB, 4-way
Texture L2 cache	128KB, 8-way
Memory configuration	1GB, 16 bytes/cycle 8 channel 8 banks per channel

the texture unit. Compared to the texture fetching latency from the memory hierarchy, the latencies for computing and accessing hash table are negligible.

VI. Evaluation Methodology

Figure 16 shows the evaluation framework which consists of two main layers: simulation layer and analysis layer. The simulation layer is responsible for generating GPU run-time information using real game traces while the analysis layer gathers the simulation results to further produce energy and image quality analysis.

Simulation layer. To evaluate PATU and its optimizations, we employ ATTILA-sim [35], a widely-adopted cycle-accurate simulator for rasterization-based GPU. It is not only highly configurable with modern 3D rendering schemes but also includes a wide spectrum of GPU hardware features. The major system simulation parameters for the baseline GPU architecture are listed in Table I. We configure the baseline architecture by referencing the PowerVR Rogue GPU [29], a state-of-the-art mobile architecture. We employ the commonly-used hardware implemented AF on commercial GPUs [13], [23], [29] as our baseline texture filtering method. Additionally, to update ATTILA-sim to reflect the newest advancements in rendering, we integrate several latest GPU hardware features into it, including tiling engine, tessellation and multi-view VR.

Analysis layer. To evaluate the power consumption of our design, we slightly modify McPAT [33] to model the basic unified shaders, caches and on-chip interconnect on GPU by using the power modeling strategy introduced in [36]. To further accurately model 3D rendering, we extend McPAT to support GPU texture memory hierarchy and calculate the power of the texture units by scaling the core power based on the number of floating-point ALUs and texture busy cycles. We also include the power modeling for GPU's DRAM based on the Micron power model [37].

To evaluate the effects on user gaming experience, frames and their rendering cycles are collected in this layer, based on which we employ MATLAB [38] to measure the image quality and build game replays. We implement vertical synchronization in MATLAB to achieve real time gaming experience based on the rendering time of the display frame. To match the static refresh rate (i.e., 60Hz) of the monitor, we draw each frame at the beginning of screen refreshing or stall the drawing process if the current frame is incomplete within the refresh interval. As a result, users may feel motion lags if the rendering time of a frame is too long to catch up with the screen refreshing. To better analyze GPU performance change, we employ a fixed CPU latency for each frame which is equal to half of the monitor refreshing interval. Therefore, the refresh interval in our experiment is 8 million cycles under 1GHz GPU frequency.

TABLE II: 3D GAMING BENCHMARKS

Abbr.	Names	Resolution	Library
HL2	Half-life 2	1600x1200 1280x1024 640x480	DirectX3D [39]
doom3	Doom3	1600x1200 1280x1024 640x480	OpenGL [32]
grid	GRID	1280x1024	DirectX3D
nfs	Need For Speed	1280x1024	DirectX3D
stal	S.T.A.L.K.E.R.: Call of Pripyat	1280x1024	DirectX3D
Ut3	Unreal Tournament 3	1280x1024	DirectX3D
wolf	Wolfenstein	640x480	DirectX3D

Benchmarking. Shown in Table II, the set of benchmarks employed to evaluate our design includes seven well-known 3D games, covering different rendering libraries and 3D engines. To analyze them, we directly use the graphics API traces from ATTILA [35] which are OpenGL and DirectX3D commands captured by the ATTILA-trace generator. For better evaluating our techniques to reflect the impact of workload size, we render two games (*Doom3 and Half-Life 2*) from the table with different resolutions (1600×1200, 1280×1024, 640×480). For the other games, we adopt 1280x1024 resolution if it is available and supported. In our experiments, all the gaming workloads run to completion on the simulator and the average rendering value for multiple frames is calculated to represent the final 3D rendering result of a game.

VII. Results And Analysis

A. Threshold Analysis: Performance-Quality Tradeoff

As we discuss in Section II-D and IV-C(C), threshold acts like a controllable knob to decide what pixels can be approximated (e.g., all but the black areas in the frame in Figure 8) after PATU predicts a pixel's SSIM index at runtime. It can be either tuned by users' experience or set to a static optimal value based on architectural design space exploration. Particularly, we can navigate design space of PATU using this threshold. Figure 17 shows the normalized speed up and quality loss of each game under different thresholds (threshold $\in$ [0,1] which is based on the numerical range of Eq.(1)). In each subfigure, we employ the MSSIM index introduced in Section II-D to evaluate the perceived quality of the rendering frame. We also normalize the overall 3D rendering performance to the baseline case with 16X AF. This figure demonstrates one of PATU's major contributions: *it creates a new performance-quality tuning space which enables targeted AF filtering on only perceivable pixels.* To conduct a fair evaluation on which threshold is the optimal for selection, we resort to a simple Speedup×MSSIM metric which puts equal weight on speedup and image quality. We

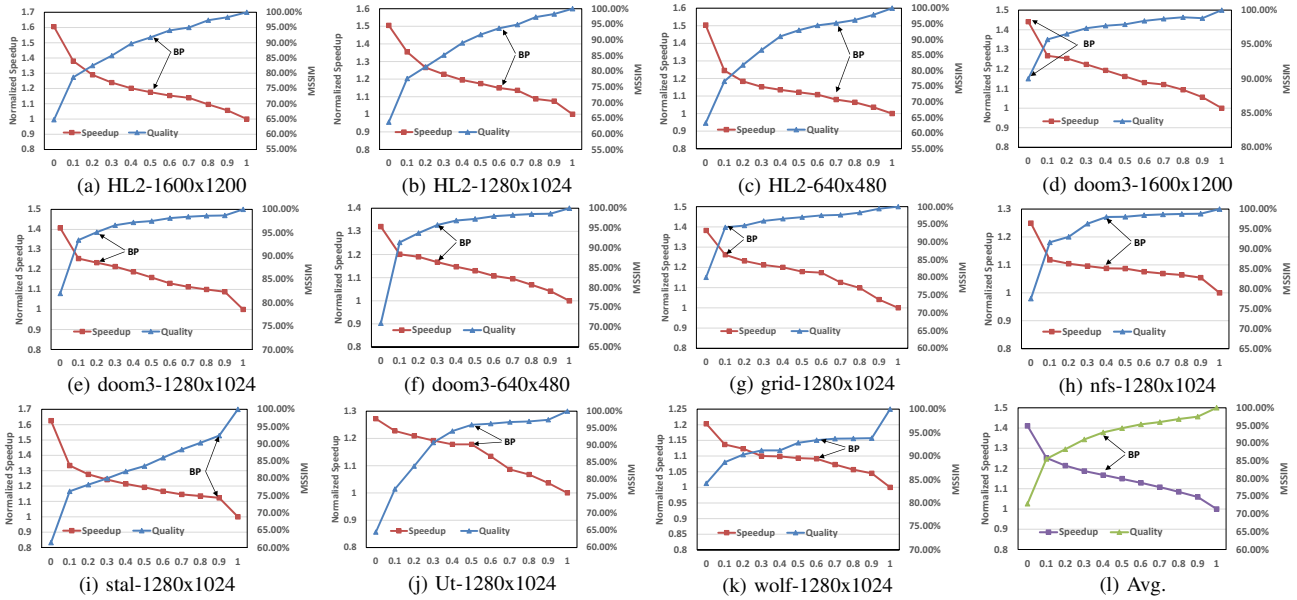

Figure 17: Threshold analysis: performance-quality trade-off across different 3D games. BP is the threshold corresponding to the highest (speedup×MSSIM) metric for the individual game. Subfigure (I) is the average case with an average BP.

denote BP (Best Point) as the threshold corresponding to the highest (speedup×MSSIM) value. Note that MSSIM is just a general approximation of user perception. For certain image frames users may very likely pick a different optimal threshold, e.g., favoring performance more than the MSSIM value since sometimes it is difficult to visually distinguish two images when *MSSIM* is higher than 90%. Without denying that, we apply this simple metric for a more fair and scientific evaluation. A complementary user experience study is also provided in Section VII-D.

We have several observations from the figure. (1) Under PATU, the tradeoff relationship between speedup and perceived quality with respect to the threshold is nearly-linear and forming an "X" shape. When the threshold value increases from the no-AF (threshold=0) to baseline (threshold=1), fewer pixels are marked as non-perceivable. As a result, the perceived quality of the frame increases while the overall 3D rendering performance decreases. (2) In almost every game, we observe a dramatic MSSIM rising when the threshold increase from 0 to 0.1. This is because PATU is able to detect these perceivable pixels which have significant impact on the overall frame quality. (3) For most games, BP is located between 0.1 to 0.9, suggesting that PATU provides better performance-quality tradeoff than the baseline or directly disabling AF. One exception is `doom3-1600x1200`, in which the quality penalty without AF is very small (only 8%). Thus directly turning off AF (threshold=0) achieves the best tradeoff. (4) We observe an interesting trend with the image resolution: the BPs for higher resolution games are smaller than those of lower resolution games (e.g., `doom3-1280x1-24` vs. `doom3-640x480`), indicating that PATU offers flexible control over the performance-quality tradeoff based on user preference. Finally, Figure 17(I) shows the average performance-quality tradeoff across all the games and their average BP threshold = 0.4. Without specific mention, we will apply this average threshold as default for the remaining of evaluation since it provides decent average speedup without visible quality loss (i.e., 94% MSSIM).

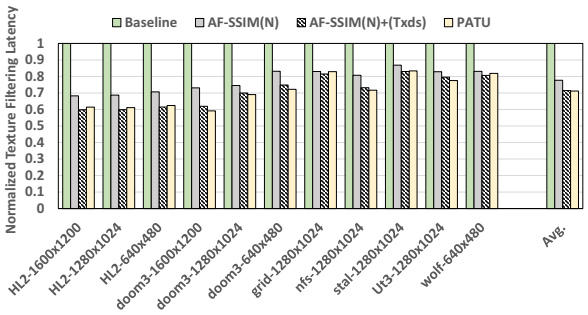

Figure 18: Normalized texture filtering latency.

B. PATU's Impact on Performance, Energy and Perceived Image Quality

(A) Performance and Image Quality Implications. We further evaluate the impact of PATU on texture filtering latency, performance/quality and energy consumption by comparing four design scenarios: (i) **Baseline**: the baseline GPU with 16xAF, (ii) **AF-SSIM(N)**: sample-area based prediction (Section IV-C), (iii) **AF-SSIM(N)+(Txds)**: sample-area based prediction+ distribution based prediction (Section IV-C); and (iv) **PATU**: two prediction methods+ LOD shift elimination (Section V). All the data are normalized to the baseline case.

Figure 18 shows the texture filtering latency reduction under different design scenarios. As discussed in Figure 5, the texture filtering latency is the major performance bottleneck for applying AF in 3D rendering. We can observe that AF-SSIM(N)+(Txds) and PATU have very similar impact on texture filtering process, both of which significantly outperforms the baseline by reducing the texture filtering latency by an average of 29% (up to 42%). They also achieve lower latency than AF-SSIM(N) due to further eliminating unnecessary AF on pixels with high texel distribution similarity. PATU not only reduces the texel requirement for

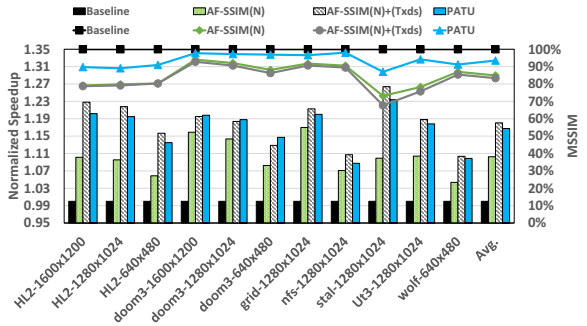

Figure 19: Normalized speedup and perceived quality of the overall 3D rendering under different design scenarios.

each pixel but also reuses high-quality texture in the texture cache to increase data locality.

3D rendering performance and rendering quality are two major effects determining users' experience. Figure 19 shows the normalized speedup of the overall 3D rendering (bars) and perceived quality measured by MSSIM (lines) for different games under four design scenarios. We have two important observations. First, we observe that AF-SSIM(N)+(Txds) outperforms all the others in terms of performance and achieves an average of 18% (up to 26%) performance improvement. However, it also loses an average of 16% perceived quality measured by MSSIM. AF-SSIM(N) only improves the overall performance by 10% than Baseline but with similar quality degradation as AF-SSIM(N)+(Txds) due to its incapability of capturing the texel distribution similarity and addressing LOD shift. On the contrary, PATU addresses the LOD shift problem and provides more than 10% of image quality improvement over AF-SSIM(N)+(Txds) with negligible performance penalty (i.e., 1.3%). It achieves an average of 93% (up to 98%) MSSIM index which is very close to the baseline (100%) and is difficult to be distinguished by human eyes. Second, PATU provides more speedup for applications with higher resolution, e.g., 7% better performance for `HL2-1600x1200` than `HL2-640x480`. This is because increasing resolution results in more pixel to be processed which also demands more texels, and PATU targets on reducing texel requirement per pixel.

In summary, under the threshold corresponding to the average highest $Speedup \times MSSIM$, PATU achieves an average of 93% (up to 98%) perceived quality and 17% (up to 24%) overall 3D rendering speedup over Baseline. In other words, PATU achieves significant performance improvement for 3D rendering without visible quality loss. Due to MSSIM's bias on only measuring the image loss by the absolute pixel value, we provide a user experience study in Section VII-D to further demonstrate that in reality users normally cannot perceive quality difference under PATU.

(B) Energy Implications. Figure 20 shows the normalized energy of the entire GPU (including DRAM) under different designs. We observe that PATU achieves an average of 11% (up to 16%) energy reduction on the entire GPU, even though it slightly increases the runtime power (about 7% on average) due to higher texel throughput. The energy savings from PATU mainly come from the significant performance improvement. In addition, PATU consumes slightly more energy (by only 1%) than AF-SSIM(N)+(Txds) because PATU fetches texels from a more detailed texture map to eliminate LOD shift, resulting in minor memory traffic increase.

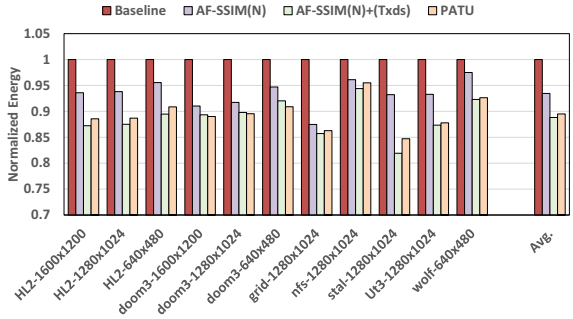

Figure 20: Normalized GPU energy under different designs.

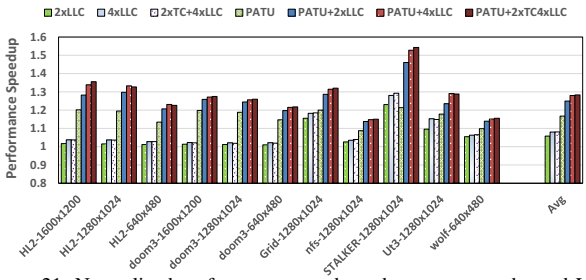

Figure 21: Normalized performance speedup when texture cache and LLC sizes are scaled up.

C. Cache Sensitive Study

To explore the effects of PATU on different cache organizations, we scale up the size of Last level cache (LLC) and Texture cache (TC). Figure 21 shows the normalized performance speedup of 3D rendering. All results are normalized to the baseline case (1x cache size without PATU). We leverage the default threshold for PATU in this experiment. We obverse that only increasing the cache capacity (2x LLC, 4xLLC and 2xTC+4xLLC) does not significantly benefit the overall 3D rendering performance while enabling PATU alongside further provides 24.1%, 28.0%, 28.3% performance speedups over baseline. Additionally, PATU scales with increasing LLC cache size. This is because for most cases texture data is loaded from GPU memory to LLC sequentially and is shared by multiple shaders. Hence, rendering performance is highly dependent on cache throughput rather than capacity. PATU significantly reduces the texel requirement and the computation for each approximated pixel, alleviating the pressure for the entire texture memory hierarchy. As a result, our technology is orthogonal to other cache optimization schemes and can provide scalable performance speedup.

D. User Experience Study

To understand how PATU impact on real users gaming experience, we rebuild the replay video for doom3 and HL2 under difference thresholds (i.e. 0, 0.2, 0.4, 0.6, 0.8). We sequentially connect 600 frames from each game, with the average video length of 28 seconds. The range of average frame rate of videos is from 33 to 58 fps, varies across applications, resolution and thresholds. The frame rate drop is due to the fact that the evaluated gaming benchmarks that are supported by our simulator are designed for high performance desktop GPUs so that not every frame can satisfy the real-time constraint. Here we use pre-recorded video generated from simulation trace, instead of letting users play live, is because our scheme requires modification to GPU architecture.

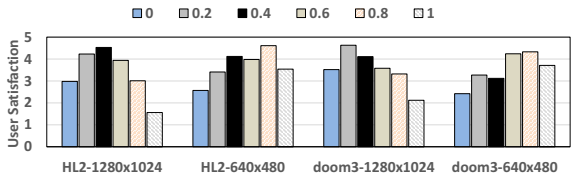

Figure 22: User satisfaction score over different thresholds.

For human inputs, we randomly recruited 30 participants on a college campus. We let them watch game videos with a random order and ask them to make evaluation using satisfaction score (i.e., 1 being the lowest and 5 being the highest). As we discussed in Section 2, although disabling AF reduces the sharpness and misses important details for the rendered image, it does not change the rendering resolution. In order to let users focus on image quality and performance change, the game replay is displayed on a unified screen size (5.5 inches) for all users. Changing screen sizes only affect perceptive pixel density (i.e., the distance between two pixels) but will not affect user perceived image quality (i.e., color and details) and rendering resolution. In other words, perceptive pixel density change will not affect users' perception on image quality and have minimal effect on user satisfaction [40]. Thus we expect our technology has similar impact on users' gaming experience for different screen sizes and platforms.

Figure 22 shows the average user satisfaction score for different thresholds. In general, we observe that PATU achieves higher scores than the baseline AF-On case (threshold=1) and no-AF case (threshold=0). Combined with Figure 17, we have some interesting observations on image resolutions. (1) For games with higher resolutions, users tend to balance performance with image quality, which PATU can effectively help them achieve. For instance, users in general prefers threshold=0.2 in `doom3-1280x1024` since it has better performance with no visible quality loss (i.e., MSSIM=95%). Decreasing threshold for high resolution applications reduces the rendering time of frames with high texture effects, thus avoiding motion lags on the screen. (2) For lower-resolution games, users put higher preference on image quality than performance. PATU also can provide effective threshold for them (i.e., 0.8 for both cases) with non-perceivable quality loss.

VIII. Related Work

Image Rendering on GPU. In recent years, 3D rendering optimization has attracted a lot of attention due to the rising market of user-interactive gaming and virtual reality. Among them, several architecture-level works [12], [19], [20], [21], [41] have been proposed. Arnau et al. [19] designed a parallel rendering model by splitting the hardware resource into multiple cluster. They also proposed a memorization mechanism [12] to reduce fragment redundancy. Gaur et al. [20] focused on GPU LLC performance and proposed a self-learning management approach for improving LLC's hit rate for 3D rendering. These works focus on exploiting either the fragment redundancy among consecutive frames or a application-specific cache design for better cache performance while ours proposes runtime approximation strategies to reduce the workload for each pixel in texture filtering based on users' perception. Thus our design is orthogonal to these proposals. Hsiao et al. [41] reduced the precision of floating-point in 3D rendering to conserve energy for the shader cores on a mobile GPU. However, we target on proposing a novel runtime approximation hardware design to reduce unnecessary AF to improve GPU performance and energy efficiency as well as maintaining user experience. The work most related to ours is [21], which replaces the GDDR5 of conventional GPU with HMC and enables its processing-in-memory capability to improve texture filtering performance. Their strategy requires replacing conventional GPU's entire memory system with 3D-stacked memory, incurring modifications on both GPU host and HMC to support a reordered execution of texture filtering. Without these constraints, our design can be generally applied to conventional GPUs on the market and requires no modification to off-chip memory. Other common ways to accelerate 3D rendering is through texture compression [8], [9], [42], [43] and dynamic resolution scaling [11], [14], which our work is also orthogonal to.

Structure Similarity. The SSIM index and its extension are widely adopted in a variety of applications, including video coding [44], image classification [45], decisioning [46] and biometrics [47]. Several works also use it as certain optimization criteria [30], [48], [49]. However, they all exploit features of the already-calculated SSIM index while our design aims to identify the similarity degree between two pixels at runtime before the actual texels are fetched from memory through texture filtering.

IX. Conclusions

In this paper, we propose a novel perception-oriented approximation hardware design for 3D rendering on GPU. First, we exploited AF's impact on 3D rendering's performance and energy consumption, memory bandwidth usage, texture filtering latency and user perception of image quality. We observed that disabling AF for a large portion of pixels may not impact user experience. Based on this insight, we exploited the inner-relationship between the anisotropic and isotropic filtering and proposed a perception-oriented approximation model by leveraging the sample area similarity and texel distribution similarity. According to this model, we further designed a low-cost perception aware texture unit that can predict AF-sensitive SSIM map at runtime and decide if a pixel can be approximated. Decision making at runtime occurs prior to the actual texture filtering. Finally, extensive evaluation on real-world 3D games demonstrated the significant benefits achieved by our design on performance, energy efficiency and texture filtering latency without sacrificing user experience. We also found that users and common design space exploration may pick different thresholds as the optimal tuning point based on factors such as resolutions.

Acknowledgment

This research is supported by U.S. DOE Office of Science, Office of Advanced Scientific Computing Research, under the CENATE project (award No. 66150). The Pacific Northwest National Laboratory is operated by Battelle for the U.S. Department of Energy under contract DE-AC05-76RL01830. This research partially supported by NSF grants CCF-1619243, CCF-1537085(CAREER), CCF-1537062

References

[1] A. Jarabo, H. Wu, J. Dorsey, H. Rushmeier, and D. Gutierrez, "Effects of approximate filtering on the appearance of bidirectional texture functions," *TCVG*, 2014.

[2] G. Knittel, A. Schilling, A. Kugler, and W. Straer, "Hardware for superior texture performance," in *EGGH*, W. Strasser, Ed., 1995.

[3] K. Boos, D. Chu, and E. Cuervo, "Flashback: Immersive virtual reality on mobile devices via rendering memoization," in *MobiSys*, 2016.

[4] "Nvdia geforce: Antialiasing and anisotropic filtering," http://www.geforce.com/whats-new/guides/aa-af-guide#1.

[5] "Bringing high-end graphics to handheld devices," https://www.nvidia.com/content/PDF/tegra_white_papers/Bringing_High-End_Graphics_to_Handheld_Devices.pdf.

[6] P. Mavridis and G. Papaioannou, "High quality elliptical texture filtering on gpu," in *I3D*, 2011.

[7] J. Pool, A. Lastra, and M. Singh, "Precision selection for energy-efficient pixel shaders," in *HPG*, 2011.

[8] S. Fenney, "Texture compression using low-frequency signal modulation," in *EGGH*, 2003.

[9] J. Ström and T. Akenine-Möller, "ipackman: High-quality, low-complexity texture compression for mobile phones," in *EGGH*, 2005.

[10] C. H. Sun, Y. M. Tsao, and S. Y. Chien, "High-quality mipmapping texture compression with alpha maps for graphics processing units," in *IEEE Transactions on Multimedia*, 2009.

[11] A. Maghazeh, U. D. Bordoloi, M. Villani, P. Eles, and Z. Peng, "Perception-aware power management for mobile games via dynamic resolution scaling," in *ICCAD*, 2015.

[12] J. M. Arnau, J. M. Parcerisa, and P. Xekalakis, "Eliminating redundant fragment shader executions on a mobile gpu via hardware memoization," in *ISCA*, 2014.

[13] I. Bratt, "The arm mali-t880 mobile gpu," in *HCS*, 2015.

[14] S. He, Y. Liu, and H. Zhou, "Optimizing smartphone power consumption through dynamic resolution scaling," in *MobiCom*, 2015.

[15] A. Shye, B. Ozisikyilmaz, A. Mallik, G. Memik, P. A. Dinda, R. P. Dick, and A. N. Choudhary, "Learning and leveraging the relationship between architecture-level measurements and individual user satisfaction," in *ISCA*, 2008.

[16] J. Park, E. Amaro, D. Mahajan, B. Thwaites, and H. Esmaeilzadeh, "Axgames: Towards crowdsourcing quality target determination in approximate computing," in *ASPLOS*, 2016.

[17] B. Thwaites, G. Pekhimenko, H. Esmaeilzadeh, A. Yazdanbakhsh, O. Mutlu, J. Park, G. Mururu, and T. Mowry, "Rollback-free value prediction with approximate loads," in *PACT*, 2014.

[18] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Boosting mobile gpu performance with a decoupled access/execute fragment processor," *ISCA*, 2012.

[19] J. M. Arnau, J. M. Parcerisa, and P. Xekalakis, "Parallel frame rendering: Trading responsiveness for energy on a mobile gpu," in *PACT*, 2013.

[20] J. Gaur, R. Srinivasan, S. Subramoney, and M. Chaudhuri, "Efficient management of last-level caches in graphics processors for 3d scene rendering workloads," in *MICRO*, 2013.

[21] C. Xie, S. L. Song, J. Wang, W. Zhang, and X. Fu, "Processing-in- memory enabled graphics processors for 3d rendering," in *HPCA*, 2017.

[22] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE Transactions on Image Processing*, 2004.

[23] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, 2008.

[24] J. P. Ewins, M. D. Waller, M. White, and P. F. Lister, "Implementing an anisotropic texture filter," *Computers & Graphics*, 2000.

[25] A. Schilling, G. Knittel, and W. Strasser, "Texram: a smart memory for texturing," *CG&A*, 1996.

[26] E. Hart, "3d textures and pixel shaders," in *Direct3D ShaderX*, 2002.

[27] M. Olano, S. Mukherjee, and A. Dorbie, "Vertex-based anisotropic texturing," in *EGGH*, 2001.

[28] "Relative benchmark," https://itunes.apple.com/us/app/relative-benchmark/id560637086?mt=8.

[29] "Powervr rogue," https://www.imgtec.com/powervr/graphics/rogue/.

[30] A. Rehman, M. Rostami, Z. Wang, D. Brunet, and E. R. Vrscay, "Ssim-inspired image restoration using sparse representation," *EURASIP Journal on Advances in Signal Processing*, 2012.

[31] D. M. Chandler, "Seven challenges in image quality assessment: past, present, and future research," *ISRN Signal Processing*, 2013.

[32] "Opengl," https://www.opengl.org/about/.

[33] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, 2009.

[34] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP Laboratories*, 2009.

[35] V. M. del Barrio, C. Gonzalez, J. Roca, A. Fernandez, and E. E, "Attila: a cycle-level execution-driven simulator for modern gpu architectures," in *ISPASS*, 2006.

[36] J. Lim, N. B. Lakshminarayana, H. Kim, W. Song, S. Yalamanchili, and W. Sung, "Power modeling for gpu architectures using mcpat," *TODAES*, 2014.

[37] "Tn-41-01: Calculating memory system power for ddr3," https://www.micron.com/resource-details/3465e69a-3616-4a69-b24d-ae459b295aae, accessed: 2016-6-24.

[38] "Matlab," https://www.mathworks.com/.

[39] "Direct3d," https://msdn.microsoft.com/en-us/library/windows/desktop/bb219837(v=vs.85).aspx.

[40] W. Zou, J. Song, and F. Yang, "Perceived image quality on mobile phones with different screen resolution," *Mobile Information Systems*, 2016.

[41] C.-C. Hsiao, S.-L. Chu, and C.-Y. Chen, "Energy-aware hybrid precision selection framework for mobile gpus," *Computers & Graphics*, 2013.

[42] J. Nystad, A. Lassen, A. Pomianowski, S. Ellis, and T. Olson, "Adaptive scalable texture compression," in *EGGH*, 2012.

[43] Y. Xiao, C.-S. Leung, P.-M. Lam, and T.-Y. Ho, "Self-organizing map-based color palette for high-dynamic range texture compression," *Neural Comput. Appl.*, 2012.

[44] A. Rehman and Z. Wang, "Ssim-inspired perceptual video coding for hevc," in *ICME*, 2012.

[45] Y. Gao, A. Rehman, and Z. Wang, "Cw-ssim based image classification," in *ICIP*, 2011.

[46] V. Bruni, D. Panella, and D. Vitulano, "Non local means image denoising using noise-adaptive ssim," in *EUSIPCO*, Aug 2015.

[47] J. Galbally, S. Marcel, and J. Fierrez, "Biometric antispoofing methods: A survey in face recognition," *IEEE Access*, 2014.

[48] D. Brunet, E. R. Vrscay, and Z. Wang, "Structural similarity-based approximation of signals and images using orthogonal bases," in *ICIAR*, 2010.

[49] D. Otero and E. R. Vrscay, "Solving optimization problems that employ structural similarity as the fidelity measure," in *IPCV*, 2014.