Processing-in-Memory Enabled Graphics Processors for 3D Rendering

Chenhao Xie*, Shuaiwen Leon Song †, Jing Wang‡, Weigong Zhang ‡, Xin Fu*
*ECE Department, University of Houston
†HPC group, Pacific Northwest National Lab (PNNL)
‡Beijing Advanced Innovation Center for Imaging Technology, College of Information Engineering, Capital Normal University
*cxie@uh.edu, †Shuaiwen.Song@pnnl.gov, ‡jwang,5591@cnu.edu.cn, *xfu8@central.uh.edu

***Abstract*—The performance of 3D rendering of Graphics Processing Unit that converts 3D vector stream into 2D frame with 3D image effects significantly impacts users gaming experience on modern computer systems. Due to its high texture throughput requirement, main memory bandwidth becomes a critical obstacle for improving the overall rendering performance. 3D-stacked memory systems such as Hybrid Memory Cube provide opportunities to significantly overcome the memory wall by directly connecting logic controllers to DRAM dies. Although recent works have shown promising improvement in performance by utilizing HMC to accelerate special-purpose applications, a critical challenge of how to effectively leverage its high internal bandwidth and computing capability in GPU for 3D rendering remains unresolved. Based on the observation that texel fetches greatly impact off-chip memory traffic, we propose two architectural designs to enable Processing-In-Memory based GPU for efficient 3D rendering. Additionally, we employ camera angles of pixels to control the performance-quality tradeoff of 3D rendering. Extensive evaluation across several real-world games demonstrates that our design can significantly improve the performance of texture filtering and 3D rendering by an average of *3.97X* (up to *6.4X*) and *43%* (up to *65%*) respectively, over the baseline GPU. Meanwhile, our design provides considerable memory traffic and energy reduction without sacrificing rendering quality.**

***Keywords*-GPU; Processing-In-Memory; 3D-Stacked Memory; Approximate Computing; 3D Rendering**

I. INTRODUCTION

To satisfy the ever-increasing user demands on gorgeous graphics and authentic gaming experience, today's game developers generally employ higher image resolutions and more color effects to render 3D frames [33]. Although modern graphics processing units (GPUs) support massive multithreading that provides high throughput for 3D rendering, they gain such throughput via issuing millions of pixels per second, putting substantial pressure on the off-chip memory. Since 3D rendering often requires highly intensive memory access, the memory bandwidth on GPUs becomes a severe performance and energy bottleneck [12], [35], [45].

Hybrid Memory Cube (HMC) [7], an emerging 3D-stacked memory technology adopting high-bandwidth interface, is one of the promising solutions for overcoming the memory wall challenge. It offers high memory capacity and bandwidth interface to the host system, and an order of magnitude higher internal bandwidth than the current DDRx systems. More importantly, its embedded logic layer offers opportunities for conducting Processing-In-Memory (PIM) which may further reduce the communication overhead between the host and memory.

Recent works [8], [17], [24], [30], [46] have shown significant performance improvement by leveraging 3D-stacked memory technologies to enable PIM on various platforms. Some of them design new accelerators based on HMC for special-purpose applications [8], [17], [30], while others exploit the benefits of PIM on general-purpose platforms [24], [46]. However, none of these solutions can be directly applied to improve the 3D rendering process on GPUs, which often consists of a comprehensive pipeline and requires significant data transmission between different hardware components. Although 3D rendering on GPU has been extensively studied [11], [20], [22], [38], [42], a knowledge gap still exists on how to leverage 3D-stacked memory to further improve its performance and energy efficiency.

Focusing on improving performance and reducing off-chip memory traffic, we observe that the memory bottleneck problem in 3D rendering is directly contributed by texel fetching in texture filtering process. Through further investigation, we identify anisotropic filtering as the most significant limiting factor for the performance of texture filtering. Based on these observations and a common assumption that leveraging the computing capability of HMC to process texture filtering in memory may reduce the data traffic caused by texel fetches, we propose a simple texture filtering in memory design, named *S-TFIM*, to directly move all the texture units from the host GPU to the logic layer of the HMC. However, the performance improvement via S-TFIM is quite trivial, a result of a considerable amount of live-texture information transmission between the host GPU and the HMC. To overcome this challenge, we propose an advanced texture filtering in memory design, named *A-TFIM*, to split the texture filtering process into two parts: performing bilinear and trilinear filtering in the host GPU while processing the most bandwidth-hungry anisotropic filtering in the logic layer of the HMC. To further reduce texel fetching, we reorder the texture filtering sequence by moving anisotropic filtering to the beginning of the filtering pipeline, while guaranteeing the correctness of the output texture. Finally, we employ a camera-angle threshold to enhance data reuse on GPU texture caches and control the performance-quality tradeoff of 3D rendering.

Contributions. We make the following contributions:

- Based on the observations from a detailed bottleneck analysis on 3d rendering, we propose two comprehensive designs, S-TFIM and A-TFIM, to leverage the

Corresponding Author: Jing Wang

2378-203X/17 $31.00 © 2017 IEEE
DOI 10.1109/HPCA.2017.37

extremely high internal bandwidth and computation capability provided by HMC for performance improvement and memory traffic reduction. To the best of our knowledge, this is the first work proposing new architectural-level designs that enables PIM-based GPU for efficient 3D rendering.

- We apply a camera-angle based threshold to enhance on-chip texel data locality and control the performance-quality tradeoff of 3D rendering.
- We evaluate our proposed designs by rendering five real-world games with different resolutions. The results show that over the baseline GPU our A-TFIM design on average (1) improves the texture filtering and overall rendering performance by **3.97X** and **43%** respectively, (2) reduces the total memory traffic by **28%**, and (3) consumes **22%** less energy. We also confirm that gaming applications with higher resolutions benefit even more significantly from our design.

II. Background And Motivation

A. 3D Rendering on GPU

3D rendering is the computer graphics process that uses 3D vertex data to create 2D images with 3D effects on a computer system. GPUs were traditionally designed as special-purpose graphics processors for 3D rendering algorithms. Fig. 1 shows the baseline GPU architecture in this study. It employs the unified shader (US) model architecture for vertex and fragment process, a popular design in modern GPUs. The 3D rendering process implemented in today's GPUs consists of three main stages: *geometry processing*, *rasterization*, and *fragment processing*. We describe the implementation of each of these stages as follows:

(1) **Geometry Processing.** During this stage, input vertexes are fetched from memory by vertex fetcher and their attributes are then computed in the unified shaders. The input vertexes are further transformed and assembled into *triangles* via primitive assembly stage, and these triangles pass through next clipping stage that removes non-visible triangles or generate sub-triangles.

(2) **Rasterizeration.** The Rasterizer processes the triangles and generates fragments, each of which is equivalent to a pixel in a 2D image. The fragments are grouped into fragment tiles which are the basic work units for the last stage of fragment processing. Note that in our baseline architecture, the Rasterizer supports tiling-based scanning and early Z test to improve cache and memory access locality.

(3) **Fragment Processing.** During this stage, fragment properties such as color and depths for each fragment are computed in the unified shader, and the frame buffer is updated with these fragment properties. Unified shaders are able to fetch extra texture data by sending texture request to texture unit for better *image fidelity*. The texture unit (the highlighted orange blocks) attached to each unified shader cluster samples and filters the requested texture data for a whole fragment tile.

B. Texture Filtering in 3D Rendering

One of most critical process in 3D rendering is *texturing filtering* (within Fragment Processing), which determines the color of 3D textures (eventually for a texture-mapped pixel [29]). The texture filtering process is deeply pipelined. After receiving a texture request, the address generator first calculates the memory address for each required texel (pixel of the texture) using triangle attributes. *Texel Fetch Unit* in the texture unit will fetch the texels. If cache hits, texture unit reads the texel data from the texture cache (L1 or L2). If not, it fetches the texel from the off-chip memory. Once all the texels of the requested texture are collected, the texture unit calculates the four-component (RGBA) color of the texture and outputs the filtered texture samples to the shader.

Generally, texture filtering computes the weighted average value of sampled texels which best approximate the correct color $C\left(x,y\right)$ of a pixel [25], as shown in (1):

$$C\left(x,y\right) = 1/M \cdot \sum_{m=1}^{M} w_s \cdot t_s. \tag{1}$$

Where w_s and t_s represent the weights and vector values of the sampled texels, respectively.

Fig. 2 shows the memory access breakdown of 3D rendering process for frames selected from several popular games (see Section VI for the detailed experimental setup). Since resolutions may affect memory access pattern, we tested different resolutions for the selected frames. The figure demonstrates that the texture fetching process in texture filtering accounts for an average of 60% of the total memory access in 3D rendering, a major contributor to the overall bandwidth usage for 3D rendering on GPU. Therefore, optimizing memory access of texture filtering, especially the fetching process, may significantly decrease the memory bandwidth requirement of 3D rendering, hence boosting the overall performance.

C. Performance Bottleneck of Texture Filtering: Anisotropic Filtering

The texture filtering process on modern GPUs commonly comprises of three steps: (1) *bilinear filtering*, (2) *trilinear filtering* and (3) *anisotropic filtering [21]*. Fig. 3 shows these three steps in a game example. Note that anisotropic filtering, a high quality texture filtering method, is applied last to enhance the sharpness of the textures on the surface that are at oblique viewing angles with respect to the camera. Built upon (1) and (2), anisotropic filtering is essential to modern gaming for eliminating aliasing in 3D rendering. Specifically, it reduces the blur and presents more details in a frame [2].

Anisotropic filtering requires an area of texels based on the position of the viewing camera to filter a particular pixel. The number of required texels in anisotropic filtering significantly increases with the level of anisotropic [31]. For example, using Elliptical Weighted Average (EWA) algorithm [31] to implement the anisotropic filtering, the maximum level of anisotropic (i.e., 16x) requires $16\times2\times4 = 128$ texels, which is 32 times of fetches required by the bilinear filtering.

Although modern GPUs employ mipmaping[1] and texture compressing to reduce texture sampling and bandwidth,

[1] Mipmaps are pre-calculated sequences of texel images, each of which is a progressively lower-resolution representation of the same image.

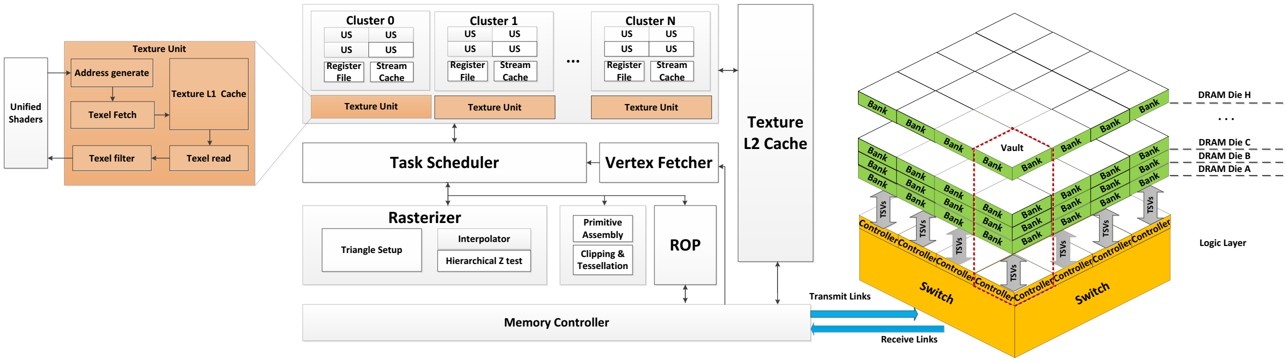

Figure. 1: The basic PIM enabled GPU design (B-PIM) with a zoom-in view on the texture unit and the connected HMC.

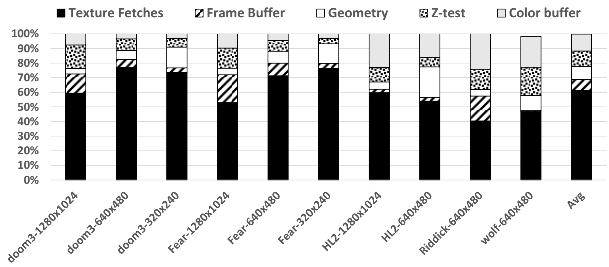

Figure. 2: Memory bandwidth usage breakdown in 3D rendering for a set of games.

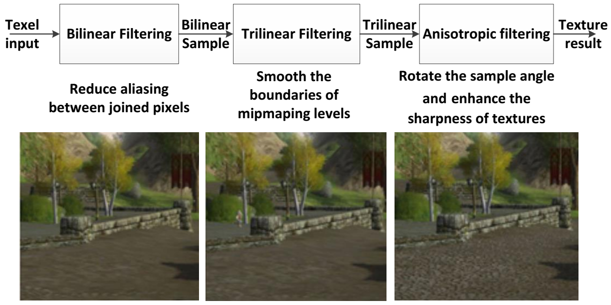
Texel input
Bilinear Filtering
Bilinear Sample
Trilinear Filtering
Trilinear Sample
Anisotropic filtering
Texture result
Reduce aliasing between joined pixels
Smooth the boundaries of mipmaping levels
Rotate the sample angle and enhance the sharpness of textures

Figure. 3: Three common steps in modern texture filtering [2].

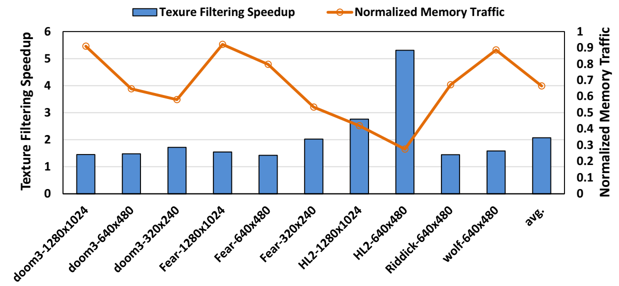

Figure. 4: Speedup of texturing filtering and texture memory traffic reduction over the baseline GPU when anisotropic filtering is disabled.

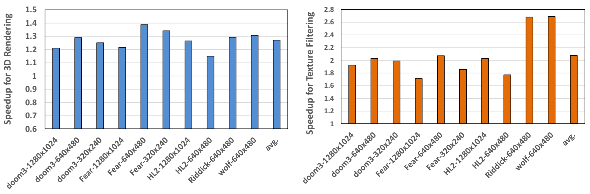

Figure. 5: Normalized speedup of 3D rendering and texture filtering using B-PIM design over the baseline.

anisotropic filtering still poses significant memory bandwidth requirement in texture filtering, directly affecting its performance. Fig. 4 shows the speedup of texture filtering and memory traffic reduction for processing five games with anisotropic filtering disabled. The performance of texture filtering increases by **1.1X** on average (up to **4.2X**), while the texture memory traffic is reduced by an average of **34%** (up to **73%**). These results clearly demonstrate that the anisotropic filtering makes up a large fraction of off-chip memory bandwidth, which is frequently the most significant limiting factor for the performance of texture filtering.

III. Basic PIM Enabled GPU For 3D Rendering

The As discussed previously, high resolution and abundant image effects significantly increase the texture requirements and memory access (e.g., anisotropic filtering) in 3D rendering on GPU, resulting in great pressure on memory bandwidth. Processing-In-Memory (PIM), which migrates data processing inside memory, has become a promising solution to accommodate high bandwidth requirements from memory-intensive applications. In this study, we consider the emerging 3D-stacked DRAM technology, i.e., Hybrid Memory Cube (HMC), as the basic implementation of PIM.

The right section of Fig. 1 shows the basic HMC architecture. In HMC, several DRAM dies are stacked on top of the CMOS logic layer, forming a cube; the cube communicates with the host GPU through a series of full-duplex I/O links. The logic layer in HMC is responsible for receiving system commands and routing memory accesses to different controllers. The controllers communicate with independent DRAM dies through a vertical set of interconnects called *Through-Silicon Vias* (TSV), which provide a vastly short interconnect path. Each controller and its sub-DRAM dies are grouped into an independent vault. A vault is similar to traditional DRAM channel since it contains a controller and several memory banks. Multiple parallel vaults and TSVs contribute to the high internal data throughput for HMC.

To evaluate the effectiveness of HMC on GPU 3D rendering, we integrate the HMC technique in Attila simu-

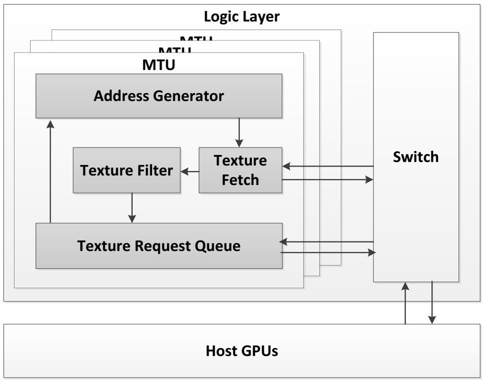

Figure. 6: Memory Texture Unit Architecture in S-TFIM.

lator [15], a cycle-accurate simulator that models modern rasterization-based GPUs. We name this implementation as basic PIM enabled GPU design (B-PIM), shown in Fig.1. We then evaluate B-PIM by rendering frames from several real games. Fig. 5 shows the normalized performance speedup of the overall 3D rendering and the texture filtering process by replacing the default GDDR5 memory with HMC. The detailed descriptions for Attila and the investigated benchmarks are shown in Section VI. The results show that B-PIM provides on average of 27% (up to 30%) and 1.07X (up to 1.69X) speed up for 3D rendering and texture filtering process respectively, over the baseline case of employing GDDR5 memory. These performance improvements are also aligned with some of the previous studies that evaluate the performance of other general-purpose applications when applying HMC [36] [8]. However, the bandwidth limitation of the off-chip links still hinders these applications from achieving further speedup: the external bandwidth is much lower than the internal bandwidth in HMC. For instance, according to the HMC 2.0 specification [7], a single HMC can provide 320 GB/s of peek external memory bandwidth via high speed serial links, while the peak internal memory bandwidth can reach to 512 GB/s per cube through 32 vaults/cube and 2 Gb/s of TSV signaling rate. A possible solution to maximize the performance of 3D rendering via HMC is to migrate the communication from external HMC (i.e., between the host GPU and HMC) to internal HMC, thus minimizing expensive off-chip memory accesses. Next we will explore design options to further optimize 3D rendering on GPU via the PIM capability in HMC.

IV. A Simple Texture Filtering In Memory Design (S-TFIM)

As described in Section II-B, texture fetching in texturing filtering process incurs intensive memory accesses and becomes the major contributor to the overall memory bandwidth usage for 3D rendering on GPU. One straightforward option is to reduce the level of texture filtering (i.e., lowering the image quality) which directly decreases the texture memory access. This has been widely adopted at algorithm level when modern 3D games started to be executed on the lower-end GPUs. However, in recent years, 3D rendering application interfaces such as OpenGL [3] and Direct3D [1] has achieved better realistic gaming experience and special effects, leading to the increasing demands from users for high-quality and expensive 3D rendering. It becomes less ideal or even intolerable to sacrifice the frame quality for performance as both of them are equally important to user experience. In other words, effectively reducing the texture memory access without losing image quality is highly desirable.

Note that the logic layer in HMC has the capability to conduct simple logic computation, and fortunately texture filtering involves relatively light calculation as shown in Eq.(1). Moreover, HMC is equipped with extremely high internal bandwidth. All these features of HMC provide great opportunities for migrating the texture filtering process into the HMC. Thus, we propose the design of Texture-Filtering-In-Memory (TFIM) that leverages the HMC internal bandwidth for high-speed texture memory access while utilizing HMC's logic layer to handle the texture filtering related computation. Through this design, the explicit texture memory accesses from the host GPU to the main memory are eliminated without sacrificing image quality.

One simple TFIM (S-TFIM) design is to directly move all the texture units from the main GPU to the HMC logic layer. We rename these texture units to *Memory Texture Units* (MTUs) in HMC. As shown in Fig. 1, each unified shader cluster contains one texture unit and there are tens of texture units in the baseline GPU. To accommodate this, we build the same amount of MTUs in the HMC to ensure that each unified shader cluster has its private MTU. One can also decrease the number of MTUs by making several shaders to share the same MTU in order to reduce the area overhead in HMC. But this design may cause resource contention for MTU and affect the overall performance of 3D rendering. Thus, to make sure that S-TFIM has the same computing capacity as the baseline, we employ private MTUs in HMC for evaluation.

Fig. 6 shows the S-TFIM design. MTUs communicate with the host GPU via the transmission (TX) and receive (RX) channels. Whenever there is a texture filtering request from the unified shader, a package is sent from the host GPU to MTU via the TX channel. This package includes the necessary information for texture filtering, such as texture coordinate information, texture request ID, and start cycle. The shader ID is also contained in the package to identify the corresponding MTU. Once arriving at the MTU, the request package is buffered into the texture request queue; in every cycle, a FIFO scheduler fetches one request to the MTU pipeline for texture filtering. Unlike the texture unit design in B-PIM (Fig. 1), a MTU in S-TFIM does not contain a texture cache since it can directly access the entire DRAM dies as its local memory. Upon completing the texture filtering, the texture data is included in a response package which is then sent back to the host GPU via the RX channel. When the texture request queue is full, MTU sends a "stall" signal to the corresponding shader which will then suspend the request package till a "resume" signal arrives.

We evaluate the effectiveness of the S-TFIM scheme and observe that the averaged performance improvement of 3D rendering across all the investigated benchmarks is trivial (only 1%) over the B-PIM design introduced in Section III. More surprisingly, S-TFIM even degrades the performance for several gaming benchmarks. After further investigation, we find that the texture requests and response packages contain a considerable amount of data that consumes much higher memory bandwidth than the normal memory read/write operations, inducing a longer delay on the package transmission.Thus the cost of transmitting the texture requests and response packages greatly offsets the performance benefits from in-memory texture filtering, causing performance degradation. This is because the host GPU of S-TFIM design no longer has L1 texture caches, losing on-chip reuse of intermediate texels (i.e., texure is the final output of texure filtering, calculated by texels. Under the baseline design, texels can be cached on-chip during filtering for future calculation of textures.). For instance, we observe the memory bandwidth usage of S-TFIM design increases by **5.37X** over B-PIM when executing $HL2$ benchmark. Therefore, a design that can selectively fetch texture requests into the HMC to maximize the PIM benefits offered by TFIM is very appealing.

V. An Advanced Texture Filtering In Memory Design (A-TFIM)

A. The Basic Idea

In this section, we propose the advanced texture filtering in memory (A-TFIM) for 3D rendering on GPU. The purpose of A-TFIM design is to drastically reduce memory access from texture fetching, which cannot be effectively addressed by B-PIM and S-TFIM. As discussed in Section II-C, texture units need to fetch all the required texels from memory before the filtering process. Shown in Fig.3 and Fig.4 from Section II-C, anisotropic filtering which often occurs after bilinear and trilinear filtering to further enhance texture sharpness, demands a large amount of texels [21] that makes the texture filtering processing extremely bandwidth-intensive.

To tackle this challenge, we decide to only move anisotropic filtering, the last step of texture filtering, to the logic layer of HMC. This decision is supported by our observation that the output of anisotropic filtering is highly reused by other filters (e.g., bilinear and trilinear filters in Fig.3) during the texturing filtering process. In other words, texture caches shown in Fig.1 can capture such texel locality and benefit the performance of other filtering phases in the same frame. This is because the added sampling area of anisotropic filtering for each texel of bilinear or trilinear filtering shares the same set of texels if the camera angle remains constant. On the contrary, the outputs from bilinear and trilinear filters are intermediate sampling results rather than texels, which are rarely reused. For example, our observation indicates that the reuse rate of trilinear results is less than 0.1% during the entire texture filtering. Thus, moving bilinear and trilinear filtering into the HMC will break the benefits of texture caches for capturing the high texel locality and may subsequently increase memory traffic.

Specifically, we disable anisotropic filtering on the host GPU as this functionality is now implemented in the HMC. However, if the design still follows the same filtering process step by step shown in Fig.3, the bilinear filter (the first phase) still requires to fetch a large number of texels as inputs in order to satisfy the demands of anisotropic filtering (the last phase). This default process is depicted in Fig.7(A). Clearly, this is suboptimal. Since bilinear filtering requires a much smaller number of input texels (discussed in Section II-C), we propose to move the anisotropic filtering phase implemented in the HMC before the bilinear filtering as the first filtering step. In this way, the texure units on the host GPU can fetch a small amount of texels from the stacked memory while the most expensive filtering is processed in memory. Fig.7(B) shows the new filtering process: under a $4X$ anisotropic filtering, the A-TFIM reduce texel fetches by $3X$ comparing to the baseline shown in Fig.7(A). The proof for the correctness of the output texture color using this new filtering sequence is shown in the next subsection.

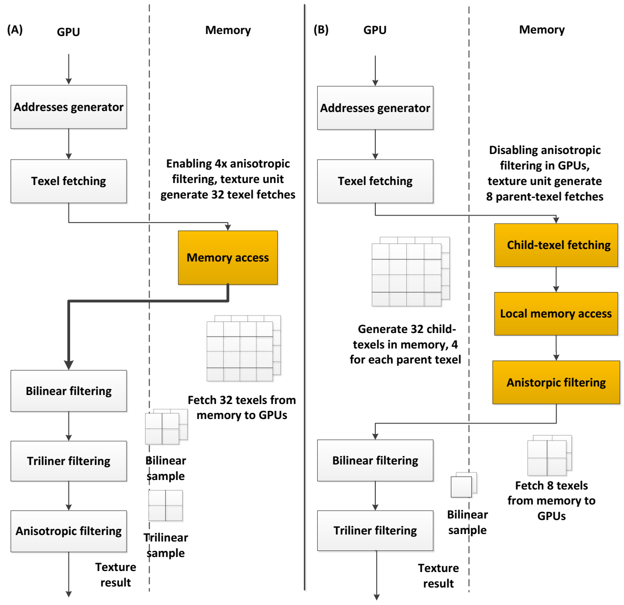

Figure. 7: (A) Baseline texture filtering fetches 32 texels from memory to GPU for a 4x anisotropic filtering. (B) A-TFIM only needs to fetch 8 texels for a 4x anisotropic filtering in HMC, where anisotropic filtering is moved in HMC and ahead of Bilinear and Trilinear filtering. This process significantly reduces memory traffic without sacrificing the correctness of texture color and image quality.

The basic flow of texture filtering in A-TFIM is as follows. First, the texture units on the host GPU fetch the required texels (i.e., the number of texels that bilinear filtering requires) from the memory stack to process texture filtering. We define these required texels with anisotropic filtering disabled as parent texels. Once the logic layer in the HMC receives the parent texel information package offloaded by the host GPU, it will generate a set of child texeles based on the texture attributes of the required parent texeles, and then feed them as inputs through the normal anisotropic filtering process in the HMC to approximate the requested parent texels. Finally, these approximated parent texels will be sent back to the texture units and cached in the texture caches as conventional inputs for bilinear and trilinear filtering. They can then be reused later in the upcoming filtering process. In this way, we not only speedup the anisotropic filtering but also reduce a significant amount of memory traffic from the

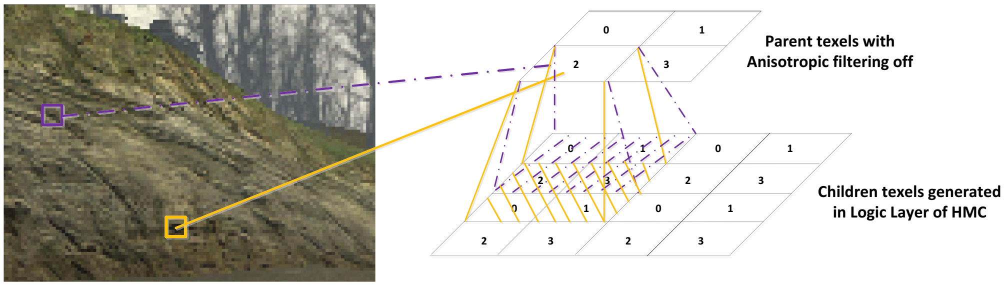
Parent texels with Anisotropic filtering off
Children texels generated in Logic Layer of HMC
0
1
2
3

Figure. 8: To filter the sunken of the stone for two pixels, a texture unit on the host GPU fetches two parent texels from them with the same address (i.e., no anisotropic filtering). Since the two pixels have different camera angles in anisotropic filtering, their corresponding parent texels should generate different sets of child texels in HMC.

texture fetching, without sacrificing the high frame quality.

B. Correctness of The New Filtering Sequence

To implement the idea, we need to guarantee the correctness of the filtering sequence reordering (i.e., moving anisotropic filtering to the beginning). In other words, the color of the output texture should remain the same after reordering. Recall Eq.(1), the general texture filtering process computes the weighted average value of the sampled texels and generates the color of texture $C(x, y)$. For example, we assume a $2\times$ anisotropic filter employing two sample texels (C_1, C_2) to calculate $C(x, y)$. Using the default sequence, the trilinear filtering first filters the samples using 4 texels ($t_1$1, $t_2$1, $t_1$2 and $t_2$2) from two mipmap levels as in Eq.(2):

$$\begin{aligned} C_1 &= t_{11} \times (1.0f - w) + t_{12} \times w \\ C_2 &= t_{21} \times (1.0f - w) + t_{22} \times w \end{aligned} \tag{2}$$

where t_i1, t_i2 are the texels from two mipmap levels, and w is the weight between two mipmap levels. Because C_1 and C_2 are from the same pixel, they have the equal weight w.

Based on Eq.(1), anisotropic filtering occurs first by simply calculating the average value of C_1 and C_2 as Eq.(3).

$$\begin{aligned} C(x, y) = &\left(C_1 + C_2\right)/2 \\ =&(t_{11} \times (1.0f - w) + t_{12} \times w \\ &+ t_{21} \times (1.0f - w) + t_{22} \times w)/2 \\ =&\,(1.0f - w) \times \left(t_{11} + t_{21}\right)/2 \\ &+ w \times \left(t_{12} + t_{22}\right)/2 \\ =&t_{1avg} \times (1.0f - w) + t_{2avg} \times w \end{aligned} \tag{3}$$

Where t_{1avg} and t_{2avg} are anisotropic filtering outputs of t_i1 and t_i2.

From this example, we find that both the final formulation of anisotropic filtering and the texel inputs are the same as those of trilinear filtering, resulting in the same ultimate C(x,y). In other words, if we first calculate the average values of t_i1 and t_i2 in anisotropic filtering, we can obtain the same color as using the original texture filtering sequence. This theory applies to both bilinear and trilinear filtering, which means we can bring the anisotropic filtering process forward as the first filtering step. The only difference is that anisotropic filtering is now providing intermediate texels like t_{1avg} and t_{2avg}, and trilinear filtering will calculate the final texture. This property is critical to the success of A-TFIM design. Our simulation results also confirm the correctness of the output texture after this reordering.

C. Accuracy Control of the Parent Texels

When the texture units on GPU receive the approximated parent texels (i.e., the output of anisotropic filtering) from the HMC, they cache them for future reuse. Occasionally, some parent texels from different frames have the same fetching address but different camera angles. For instance, Fig.8 shows that both pixels (marked in orange and purple) are colored as the sunken of the stone. In A-TFIM, a texture unit fetches the two parent texels with the same address. However, these two requested parent texels should have different set of child texels (e.g., the orange and purple shadow areas) for anisotropic filtering in the HMC because they are required by two different pixels with different camera angles. Directly reusing the same parent texel from different pixels may cause the pixel color inaccurate. Note that all the parent texels from the same pixel share the same camera angle.

We propose a simple mechanism to solve this problem. As mentioned, fetching address and camera angle determine the set of children texels for a specific parent texel. When the texture units successfully fetch the requested parent texels from the HMC, they will also store the angles of the surface pixels in the texture caches. When a texture unit fetches a new texel and receives cache hit, it compares the surface-angle of the current texel with the stored surface-angle. If the difference is smaller than a angle-threshold, the texture unit reuses the texel data from cache directly. Otherwise, the texture unit treats the texel fetch as a miss and re-fetch from the HMC so that the parent texel can be recalculated to ensure accuracy. To control the accuracy-performance tradeoff, the angle-threshold can be configured. A lower threshold often means a higher recalculation rate and a higher texture filtering accuracy. In Section VII-D, we further investigate the performance and accuracy impact from the angle-threshold.

D. Structure of A-TFIM

Fig.9 presents the high-level architectural diagram of our A-TFIM design. The implementation of the anisotropic filtering on the HMC side consists of four components: ❶ *Texel Generator* to calculate the addresses of child texels, ❷ *Child Texel Consolidation* to combine the child texel fetches,

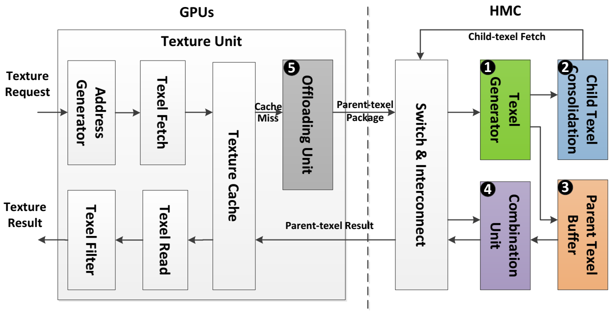

Figure. 9: Architecture Diagram for A-TFIM.

❸ *Parent Texel Buffer* to store the in-processing parent texel information, and ❹ *Combination Unit* to collect results of the child texel fetches and approximate the values of the requested parent texels. We will introduce each of them and explain how they support our proposal.

The Texel Generator receives the parent-texel fetches from the logic layer of the HMC. Then, it generates the child texels and calculates their memory addresses. After that, it reserves an entry in the Combination Unit to store the calculated parent texels.

The Child Texel Consolidation merges the identical child texel fetches to reduce memory contention. It attaches each of the parent-texel ID to the merged child-texel fetches. It then generates memory reads for the merged child-texel fetches, and sends these reads to the switch to fetch the child texel data from the HMC.

The Parent Texel Buffer is responsible for holding the in-processing parent texel information. Its entries can be accessed by the Combination Unit to update the calculated values of the parent texels. In our design, we set the buffer size to be 256 which is equal to the size of the memory request queue to avoid data loss.

The Combination Unit receives the fetched child texels from the switch and then updates the values of the parent texels. For a child texel that belongs to multiple parent texels (Section V-C), Combination Unit accesses the entries of Parent Texel Buffer to make sure all the parent texels are updated. After the parent texels are filtered completely, the Combination Unit outputs them to the switch as a normal texel fetch result back to the GPU texture units.

There are 16 texel address ALUs in the Texel Generator and 16 filtering ALUs in the Combination Unit to support parallel filtering of multiple parent texels in a 16x anisotropic filter. The system is pipelined to support high throughput texel stream. In addition, to reduce the size of the fetching package, the Offloading Unit (❺) employs a hash table to pair each of the parent texels with its offset to the first parent texel's address. In the logic layer of the HMC, we add a decomposing stage at the beginning of the Texel Generator to access the hash table and regenerate the parent texels' addresses. When the anisotropic filtering process is completed, a composing stage is added at the end of the Combination Unit to group the requested parent texels together to ensure the output package has the same format as a normal bilinear fetch. Since the aligned texels from the same fetch request share the same camera angle, we store one camera angle in each texture cache line to reduce the storage overhead. A detailed design overhead analysis will be provided in Section VII-E.

E. Walkthrough of the Texture Filtering in A-TFIM

As shown in Fig.7(B), The texture filtering process of A-TFIM can be split into two parts, *Parent-Texel Filtering* in the texture units on the host GPU and *Child-Texel Filtering* in the logic layer of the HMC. After receiving texture request, a texture unit first calculates the memory addresses of the requested parent texels as if anisotropic filtering is disabled. Next, it fetches parent texels from the texture caches. Upon a hit, it further compares the camera angle of a parent texel with the cached camera angle. If the difference is greater than the angle-threshold, it treats the fetch as a miss. Otherwise, it directly fetches the texel data from cache. Upon a miss, the Offloading Unit packs the parent-texel info and sent it to the HMC through the transmit links.

When the switch detects the arrival of a parent-texel fetching package from the link, it passes the package to the Texel Generator. The Texel Generator calculates the coordinates of child texels using the packed parent texel information and pixel coordinates. Then it calculates their addresses and generates child texel fetches. The parent texel ID info is added to the child texels to identify their associated parent texels. After generating all the child texels for the requested parent texels, the Texel Generator stores the parent texels' info in the Parent Texel Buffer and sends these child texel fetches to the Combination Unit, which then merges the child texel fetches and marks the child texels that are required by multiple parent texels. After the switch receives child-texel reads, it routs the memory accesses to the corresponding vaults and waiting for response. Finally, it receives the fetched child texels from the switch and detects their associated parent texel IDs. After all the child texels are fetched, the requested parent texels are calculated and sent back to the host GPU for further filtering.

The texture units of the host GPU treats the responded parent texels from the HMC as normal fetch results to feed into the bilinear filter. Meanwhile, they also cache the camera angles of these parent texels. After all the required parent texels are fetched, the texture units process bilinear and trilinear filtering and output the texture results to the shaders. Note that under the scenario of multiple HMCs connected to one GPU [7], a parent texel fetch package from a texture unit will be mapped to a single HMC because the requested parent texels and their generated child texels access to different mipmap levels of the same texture.

VI. Evaluation Methodology

Simulation Environment. We evaluate the impact of our proposed designs on 3D rendering by modifying ATTILA-sim [15], a cycle-accurate rasterization-based GPU simulator which covers a wide spectrum of hardware features in modern GPUs. The model of ATTILA-sim is designed upon *boxes* (a module of functional pipeline) and *signals* (simulating the interconnect of different components). It is also highly configurable with the emerging 3D-rendering technologies employed by the modern GPUs. We integrate

TABLE. I: SIMULATOR CONFIGURATION

Host GPU	
Number of cluster	16
Unified shader per cluster	16
Unified shader configuration	simd4-scale ALUs 4 shader elements 16x16 tile size
GPU frequency	1GHz
Number of GPU Texture Units	16 for baseline 0 for S-TFIM 16 for A-TFIM
Texture unit configuration	4 address ALUs 8 filtering ALUs
Texture L1 cache	16KB, 16-way
Texture L2 cache	128KB, 16-way
Memory	
Off-chip bandwidth	128GB/s for GDDR5 320 GB/s total for HMC
Memory frequency	1.25 GHz for GDDR5 1.25 GHz for HMC
HMC configuration	32 vaults, 8 banks/vault 1 cycle TSV latency [13]
S-TFIM	
Number of MTU	16
MTU configuration	4 address ALUs 8 filtering ALUs
A-TFIM	
Texel Generator	16 address ALUs
Combination Unit	16 filtering ALUs

ATTILA-sim with a HMC block, which is modeled based on multiple released sources including HMC-sim [26] (modeling a low-latency memory stack and its logic layers), the performance evaluation of HMC by Rosenfeld et al. [36], the latency evaluation of HMC by Chen et al. [13], and a commercial HMC specification [7]. The off-chip links between the host GPU and the HMC are modeled as full-duplex signal channels. The transmit latency and bandwidth of channels are adjusted to simulate high-speed serial links. We model the size of an offloading package as $4X$ the size of a normal memory read request package, and the size of a response package of TFIM is equal to the size of a read response package from HMC. We also model the access latencies of the added structures in the HMC based on those in the baseline texture units [23]. Table I shows the major system simulation parameters for different designs in this paper. Our baseline configuration is similar to the AMD TeraScale2 architecture [23], a state-of-the-art rasterization-based GPU architecture. For fairness, each design has the same number of texture units per cluster.

Energy Consumption Estimation. To evaluate energy consumption, we employ McPAT [27] to model power for the unified shaders, caches and on-chip interconnect on GPU. We slightly modify McPAT to model texture, Z-test and color caches (Z-test and color caches are hidden in ROP in Fig. 1), and to support simd4-scalar ALU. We evaluate the dynamic power of the texture units (both GPU texture units and S-TFIM's MTUs) and the computing units in A-TFIM by scaling the shader ALU power based on the number of floating-point ALUs and texture busy cycles. Additionally, we estimate their leakage power by adding 10% extra power in total, similar to the strategy used in [28]. Furthermore, we apply similar approaches proposed by K. Hsieh et al. [24] to evaluate the energy consumption of the HMC through links, TVS and DRAM power. We assume the links consume 5pJ/bit [6] while DRAM consumes 4pJ/bit. The power for the baseline GDDR5 memory is estimated by the Micron power model [4].

TABLE. II: GAMING BENCHMARKS

Names	resolution	library	3D engine
Doom3	1280x1024 640x480 320x240	OpenGL	Id Tech 4
Fear	1280x1024 640x480 320x240	D3D	Jupiter EX
Half-life 2	1280x1024 640x480	D3D	Source Engine
Riddick	640x480	OpenGL	In-House Engine
Wolfenstein	640x480	D3D	Id Tech 4

Evaluated Applications. We evaluate five well-known 3D games covering different rendering libraries and 3D engines. We directly employ the graphic traces from ATTILA [15] which represent the OpenGL and D3D commands from the CPU. These traces are captured by the ATTILA-trace generator. Additionally, we render three games (*Doom3, Fear, Half-life 2*) with different resolutions. Table I summarizes the evaluated applications. In our experiments, all the workloads run to completion on the simulator.

VII. Results And Analysis

To evaluate the effectiveness of our proposed A-TFIM design, we compare it with three other design scenarios: (i) **Baseline** — baseline GPU with GDDR5 as main memory, (ii) **B-PIM** — basic PIM enabled GPU design with a HMC as off-chip memory (Section III), and (iii) **S-TFIM** — moving all the texture units from the host GPU to the HMC logic layer (Section IV). We provide results and detailed analysis of our proposed design on performance, memory traffic, energy consumption and performance-quality tradeoff. We close this section with design overhead analysis.

A. Impact on Performance

Fig.10 shows the normalized speedup of the texture filtering process under four design options. We count the latency for texture filtering from the time when a shader sends out the texel fetching request to when it receives the final texture output for completing the filtering process (i.e., going through all three filters). All the data points are normalized to the baseline case. For A-TFIM, we set the camera angle threshold as 1.8 degree (0.01π), which is a relatively high accuracy criteria. We have two main observations from this figure. First, our proposed A-TFIM design significantly outperforms all the other three designs. For instance, A-TFIM increases the texture filtering speed by $3.97X$ (up to $6.4X$) over the baseline. A-TFIM not only leverages the high internal bandwidth provided by the HMC to accelerate the memory-intensive anisotropic filtering but also reduces significant memory traffic through reorganizing the filtering sequence. Second, A-TFIM provides more filtering speedup for gaming applications with higher resolutions. This is because these games usually demand higher anisotropic level and texel details, which is the ideal optimization target for A-TFIM design.

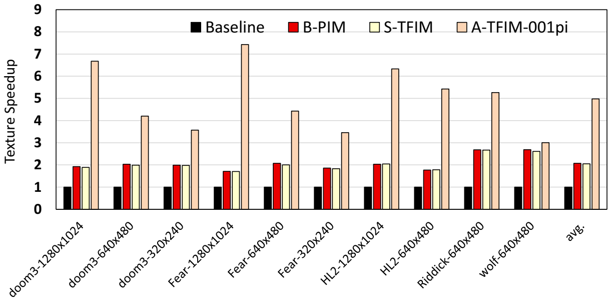

Figure. 10: Normalized speedup of texture filtering under different designs.

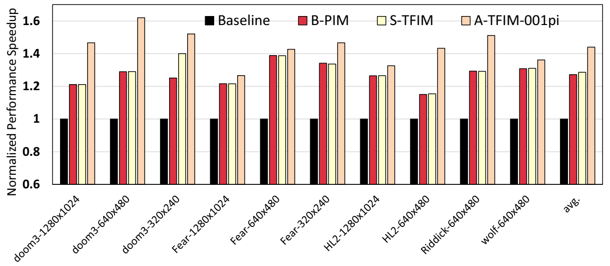

Figure. 11: Normalized speedup of 3D rendering under different designs.

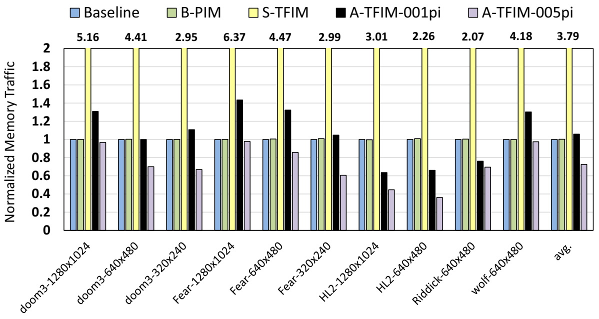

Figure. 12: Texture memory traffic induced by different designs.

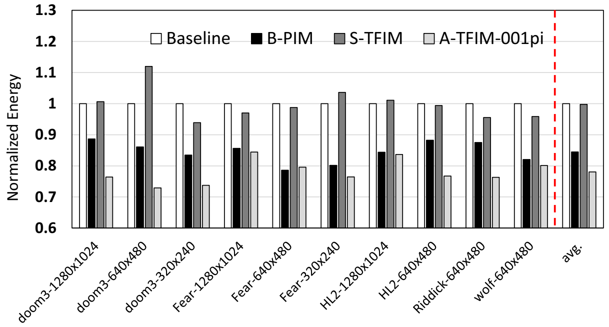

Figure. 13: Normalized energy consumption under different designs.

Fig.11 demonstrates how different designs impact the overall 3D rendering performance. A-TFIM still outperforms all the other designs in terms of overall 3D rendering performance and achieves an average of 43% (up to 65%) speedup over the baseline. Compared to A-TFIM, both B-PIM and S-TFIM suffer from some design deficiencies and exhibit similar performance. B-PIM is unable to leverage the high internal bandwidth and computing capability offered by the HMC, while S-TFIM is bottlenecked by the memory traffic due to the lack of on-chip texel caching ability for reuse. For example, among all the applications, S-TFIM only provides higher performance than B-PIM for doom3-320x240 because of its small number of texture requests.

B. Impact on Memory Traffic

The total amount of off-chip texture memory traffic is another important criteria for evaluating the effectiveness of A-TFIM. Fig.12 shows the impact of A-TFIM on the off-chip texture memory traffic, which is measured by the total transmit bytes of the texture requests between the host GPU and the memory system (either GDDR5 or HMC) during texture filtering. We only show the texture memory traffic here because our proposed design has insignificant impact on other system components.

There are two major takeaways from this figure. First, A-TFIM with a less strict camera-angle threshold (i.e., 9 degree (0.05π)) is the more effective in reducing the overall texture memory traffic, by an average of 28% (up to 64%) over the baseline. However, when the camera-angle threshold becomes more strict (i.e., 1.8 degree (0.01π)), A-TFIM slightly increases the texture memory traffic over the baseline, even though disabling anisotropic filtering on GPU in A-TFIM has reduced the total amount of required texel fetches. There are two main reasons behind this: (1) The parent texel packages offloaded by the texture units in A-TFIM may increase the texture memory traffic for some applications. (2) A more significant factor is the recalculation of the parent texels for higher rendering accuracy (Section V-C), which increases texture cache miss rate. We can control the recalculation rate by configuring the camera-angle threshold to balance the performance-accuracy tradeoff. We further discuss this in Section VII-D. Although A-TFIM slightly increases the total texture memory traffic under a higher accuracy criteria, we believe the design is still very effective considering the high performance benefit shown in Fig.10. Second, S-TFIM design, which directly moves all the texture filtering to the HMC, increases the texture memory traffic by an average of $2.79X$ over the baseline. This proves that a more comprehensive strategy like A-TFIM is needed to truly enable efficient PIM for 3D rendering on GPU.

C. Energy Consumption

Fig.13 shows the normalized energy consumption of the entire GPU (including any newly added components) under different designs. We make the following observations. First, A-TFIM (with 1.8 degree camera angle) provides the best energy efficiency among all the designs. On average, it consumes 22% less energy compared to the baseline and 8% compared to B-PIM. With further investigation, the energy savings from A-TFIM mainly come from the significant performance improvement, even though A-TFIM requires a higher average power than the others. Second, the comparison between the baseline and B-PIM indicates that HMC is more energy efficient than GDDR5. The reason is that HMC decreases the length of the electrical connections between memory controllers and DRAM devices [36]. Third, S-TFIM

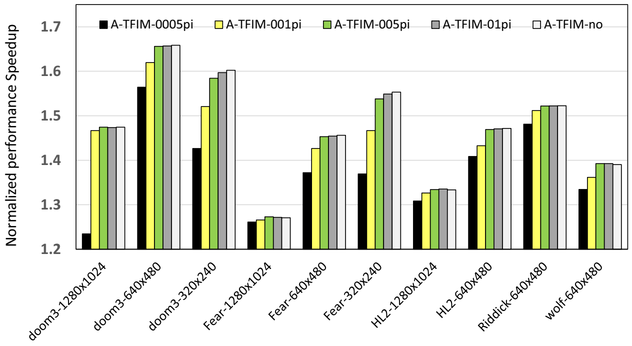

Figure. 14: Normalized performance speedup of A-TFIM under different camera-angle thresholds.

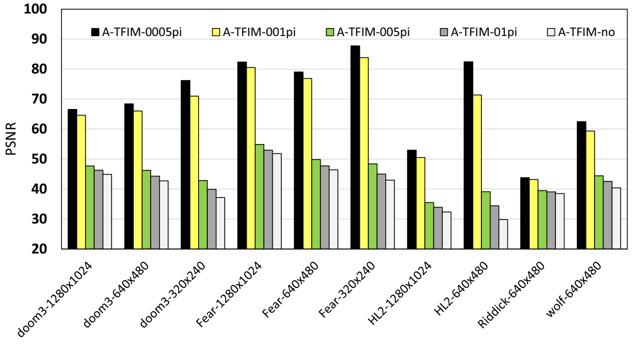

Figure. 15: Image quality (PSNR) of A-TFIM under different camera-angle thresholds.

consumes more energy than B-PIM. This is mainly due to the extra texture memory traffic shown in Fig.12.

D. *Performance-Quality Tradeoff*

As discussed in Section V-C, we can control the 3D rendering quality by configuring the camera-angle threshold. To quantify the rendering quality difference, we compare the rendered frame from A-TFIM to the output of the baseline, and detect the quality loss by calculating the *Peak Signal-to-Noise Ratio* (*PSNR*) of the rendering frames [37]. PSNR is the most commonly used metric to measure the quality of image reconstruction on GPU [14], [32]. Many works [5], [16], [34] have argued that PSNR is more sensitive than *Structure Similarity* (*SSIM*) [41] for evaluating high-quality images. A higher PSNR generally indicates higher quality. Note that the PSNR of the baseline is 99 (comparing two identical images) and when the reconstruction quality is over 70 in PSNR, users can hardly perceive the difference between two images.

Fig.14 and Fig.15 show the normalized performance speedup (normalized to the baseline) and quality loss under different camera-angle thresholds. We configure the threshold from *no recalculation* (least strict) to 0.9 degree (0.005π) (i.e., 0.005π is applied here as the most strict angle for this study because further decreasing it will cause significant performance degradation with little perceivable image quality improvement.). Note that the configuration with anisotropic filter disabled (i.e., only Isotropic) has even lower PSNR than A-TFIM-no-recalculation. These two figures reflect the expectation that in general when the value of the camera-angle threshold decreases (indicating a more strict criteria for parent texel reuse in texture caches), the quality of the frame increases while the 3D rendering performance decreases. To further explore the impact of camera-angle threshold on performance-quality tradeoff and identify the optimal threshold for the A-TFIM design, we average the speedup and PNSR values across all the applications and illustrate the performance-quality tradeoff, shown in Fig.16. It clearly shows that a smaller threshold increases the image quality while reducing performance improvement. Also, for our evaluated applications, there is a dramatic decrease of image quality when we increase the threshold from 1.8 degree (0.01π) to 9 degree (0.05π). This is because our tile-based rendering cannot detect the difference between the textures if the camera angle threshold is greater than 9 degree. In other words, it cannot identify the parent texels that should be recalculated. Furthermore, when the threshold is less than 1.8 degree (0.01π), A-TFIM will constantly recalculate parent texels which fails to reuse texel data in texture caches and unnecessarily increases the texture memory traffic without even increasing the image quality much. Therefore, we set 1.8 degree (0.01π) as our default threshold for our A-TFIM design.

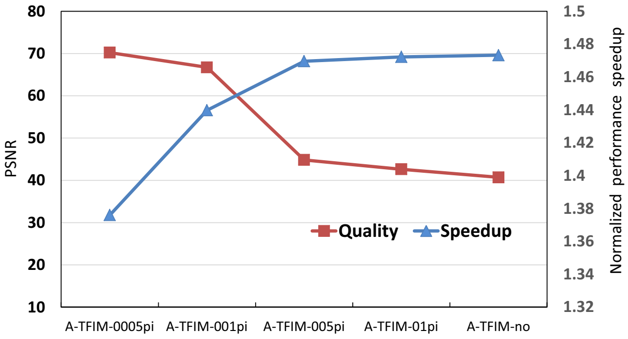

Figure. 16: Performance-Quality Trade-off.

E. *Design Overhead Analysis*

The major overhead of our A-TFIM design comes from three components: (1) the storage required for the Parent Texel Buffer and the Child Texel Consolidation in the logic layer of HMC (Fig. 9), (2) the logic units to generate child texels (Texel Generator) and approximate the value of parent texels (Combination Unit) in HMC (Fig. 9), and (3) the extra bits to record the camera angles of parent texels in texture caches on GPU. We use CACTI 6.5 [43] and McPAT [27] to estimate the area overhead of added storage and logic structures under 28nm technology.

HMC Side. For the Parent Texel Buffer, each entry contains the parent texel ID (8 bits), the temporary value of parent texel (32 bits), 1 bit to indicate whether the parent texel is filtered completely and 4 bits to count the unfetched child texels. With 256 entires, it requires $(256 \times 45)/(1024 \times 8) = 1.41$KB in the logic layer of HMC. For the Child Texel Consolidation, we add a 256-entries buffer to hold the child-parent pair ID which requires 0.5KB. To estimate the cost of the two logic units (i.e., Texel Generator and Combination Unit), we refer to a normal texture unit in Nvidia Tesla architecture [29]. The ALU arrays of the Texel Generator and Combination Unit employ floating-point vector ALUs with width 16. In summary, we estimate the area overhead of the logic units as 6.09 mm^2 and the storage buffers as 1.12 mm^2. The total area overhead of the logic layer in HMC is 3.18% of an 8Gb ($\sim 226.1mm^2$ [39]) DRAM die area.

Host GPU. To record the camera angles with high accuracy as 1^o ($180^o/2^7$), we add an extra 7 bits to each texture cache line. For a 16KB L1 texture cache with 64B cache line and a 128KB L2 texture cache with 64B cache line, we need $(250 \times 7)/(1024 \times 8)$ = 0.21 KB per L1 texture cache and $(2,000 \times 7)/(1024 \times 8)$= 1.75 KB per L2 texture cache. After adding them to the baseline configuration with 16 texture units, the total storage overhead for the host GPU in A-TFIM is 4.2 KB, and the total area overhead is 0.31 mm^2 which is only 0.23% of the entire GPU area ($136.7mm^2$). Note that an offloading unit is also added to the host GPU but it only contains a simple control logic which incurs negligible hardware overhead compared to the others above.

VIII. Related work

Processing-In-Memory on 3D stacking. With the advancement of 3D-stacked memory, several proposals [8], [17], [24], [30], [46] have studied processing-in-memory through integrating computing units in memory. Zhang et al. [46] proposed to integrate GPGPU streaming multiprocessors (SMs) into 3D-stacked memory. They also introduced a methodology to predict the performance and energy consumption for their proposed architecture. However, their approach lacks the consideration of data movement between GPU SMs and memory. Moreover, they did not discuss the types of computing that can be offloaded to memory. Hsieh et al. [24] proposed a compiler method that identifies offloading operations by calculating the bandwidth benefit and programmer-transparent data mapping to address the communication problem between multiple memory devices. To implement this idea, they introduced a hardware design to control and monitor the offloading process. Different from theirs, our work focuses on how to effectively leverage the high internal bandwidth of the HMC in 3D rendering without increasing the texture memory traffic and sacrificing the image quality. Some other works [8], [17], [30] propose to leverage 3D-stacked memory to create separate accelerators for certain types of applications (e.g., parallel graph applications [8]) in order to enable fast near-data processing. However, these PIM-based customized accelerators may not be effective for 3D rendering, which is traditionally accelerated on GPU. Instead of creating a new accelerator, our design creates a combined effort from HMC and rasterization-based GPU to customize a highly efficient solution for 3D rendering, benefiting from the best of both worlds. Additionally, we only selectively enable one filtering phase on HMC based on its unique application characteristic to achieve significant performance benefit for texture filtering.

GPU Image Rendering. GPU image rendering has been extensively studied since GPU was released to serve graphics processing [11], [20], [22], [38], [42]. With the advancement of hardware technology, the majority of these works became less effective to address the high performance and image quality demands from modern gaming. Among them, the most relevant work to ours is Schilling et al.'s proposal [38], which integrates texture computing in logic-embedded memory. However, as we discussed in Section IV, directly moving all the texture units to the HMC without considering on-chip texel data reuse will significantly increase texture memory traffic due to the large number of texture requests from the Shaders. In recent years, several works have been proposed to optimize 3D rendering on GPU. Gaur et al. [19] proposed a self-learning cache management policy for GPU's LLC to increase the hit rate and improve overall performance. Arnau et al. [9], [10] split a GPU SMs into two parts where two consecutive frames are rendered in parallel. They also proposed a memorization fragment unit to reduce memory access. These works propose new techniques to accelerate 3D rendering on GPU while our work leverages the high internal bandwidth and in-memory computing capability provided by the HMC to improve texture filtering performance. To the best of our knowledge, our work is the first architecture design enabling PIM-based GPU for 3D rendering.

Texture Compression. One common way to reduce texture memory traffic is through texture compression [**?**], [18], [32], [40], [44]. For example, one of the widely used texture compression methods supported by modern GPUs is Adaptive Scalable Texture Compression (ASTC) [32], which is a fixed-rate, lossy texture compression system. Our work is orthogonal to these texture compression techniques.

IX. Conclusions

In this paper, we enable processing-in-memory based GPU for efficient 3D rendering. First, we implement a basic PIM scheme for GPUs by integrating HMC. Then, we design a simple approach (S-TFIM) that directly moves all texture units of the host GPU into the logic layer of the HMC, leveraging the internal memory bandwidth of the HMC for texture filtering but increasing unnecessary memory traffic for data movement. To address this issue, we propose an advanced design (A-TFIM) that reorders the texture filtering sequence and precalculates the anisotropic filtering for each fetched texel in the logic layer of HMC. Finally, we propose an approximation scheme to control the performance-quality tradeoff to accompany A-TFIM. We demonstrate that A-TFIM provides significant performance speedup, memory traffic reduction and energy conservation. Meanwhile, the approximation scheme can successfully navigate the tradeoff between image quality and performance under A-TFIM. We hope our PIM-based scheme for 3D rendering on GPU can shed a light on future gaming hardware design using emerging technologies.

X. Acknowledgments

This research is partially supported by the U.S. DOE Office of Science, Office of Advanced Scientific Computing Research, under award 66150: "CENATE - Center for Advanced Architecture Evaluation". The Pacific Northwest National Laboratory is operated by Battelle for the U.S. Department of Energy under contract DE-AC05-76RL01830. This research is partially supported by NSF grants CCF-1619243, CCF-1537085(CAREER), CCF-1537062, NSFC grant No.61402302, and NSFC No.61472260.

REFERENCES

[1] "Direct3d," https://msdn.microsoft.com/en-us/library/windows/desktop/bb219837(v=vs.85).aspx, accessed: 2016-6-14.

[2] "Nvdia geforce: Antialiasing and anisotropic filtering," http://www.geforce.com/whats-new/guides/aa-af-guide#1, accessed: 2016-6-18.

[3] "Opengl," https://www.opengl.org/about/, accessed: 2016-6-13.

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

[5] "Video quality characterization techniques," http://hirntier.blogspot.com/2010/01/video-quality-characterization.html.

[6] "Initial hybrid memory cube short-reach interconnect specification issued to consortium adopters," 2012, denali Memory Report.

[7] "Hybrid memory cube specification 2.0," in *Techo. Rep.*, Nov. 2014.

[8] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *ISCA*, June 2015.

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

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

[11] A. C. Barkans, "High quality rendering using the talisman architecture," in *HWWS*, 1997.

[12] A. R. Brodtkorb, T. R. Hagen, and M. L. SæTra, "Graphics processing unit (gpu) programming strategies and trends in gpu computing," in *J. Parallel Distrib. Comput.*, Jan. 2013.

[13] K. Chen, S. Li, N. Muralimanohar, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "Cacti-3dd: Architecture-level modeling for 3d die-stacked dram main memory," in *DATE*, 2012.

[14] Y.-J. Chen, C.-H. Hsu, C.-Y. Hung, C.-M. Chang, S.-Y. Chuang, L.-G. Chen, and S.-Y. Chien, "A 130.3 mw 16-core mobile gpu with power-aware pixel approximation techniques," in *JSSC*, 2015.

[15] 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*, March 2006.

[16] R. Dosselmann and X. D. Yang, "A comprehensive assessment of the structural similarity index," in *SIVP*, 2011.

[17] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules," in *HPCA*, Feb 2015.

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

[19] 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.

[20] Z. S. Hakura and A. Gupta, "The design and analysis of a cache architecture for texture mapping," in *ISCA*, 1997.

[21] E. Hart, "3d textures and pixel shaders," in *Direct3D ShaderX*, 2002, p. 428.

[22] J. Hasselgren and T. Akenine-Möller, "An efficient multi-view rasterization architecture," in *EGSR*, 2006.

[23] M. Houston, "Anatomy of amd terascale graphics engine," in *SIGGRAPH*, 2008.

[24] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. OConnor, N. Vijaykumar, O. Mutlu, and S. W. Keckler, "Transparent offloading and mapping (tom): Enabling programmer-transparent near-data processing in gpu systems," in *ISCA*, 2016.

[25] G. Knittel, A. Schilling, A. Kugler, and W. Straer, "Hardware for Superior Texture Performance," in *EGGH*, 1995.

[26] J. D. Leidel and Y. Chen, "Hmc-sim: A simulation framework for hybrid memory cube devices," in *IPDPSW*, May 2014.

[27] 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.

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

[29] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," in *MICRO*, March 2008.

[30] G. H. Loh, N. Jayasena, M. Oskin, M. Nutter, D. Roberts, M. Meswani, D. P. Zhang, and M. Ignatowski, "A processing-in-memory taxonomy and a case for studying fixed-function pim," in *WoNDP*, 2013.

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

[32] J. Nystad, A. Lassen, A. Pomianowski, S. Ellis, and T. Olson, "Adaptive scalable texture compression," in *SIGGRAPH/EGGH-HPG*, 2012.

[33] A. Rege, "An introduction to modern gpu architecture," in *En ligne]*, 2008.

[34] A. R. Reibman and D. Poole, "Characterizing packet-loss impairments in compressed video." in *ICIP*.

[35] P. Rogers and C. FELLOW, "Amd heterogeneous uniform memory access," in *AMD Whitepaper*, 2013.

[36] P. Rosenfeld and et al., "Peering over the memory wall: Design space and performance analysis of the hybrid memory cube," Technical Report UMD-SCA, 2012.

[37] A. N. S, "Program for image / picture quality measures calculation," https://www.mathworks.com/matlabcentral/fileexchange/25005-image-quality-measures, 2012.

[38] A. Schilling, G. Knittel, and W. Strasser, "Texram: a smart memory for texturing," in *IEEE Computer Graphics and Applications*, May 1996.

[39] M. Shevgoor, J.-S. Kim, N. Chatterjee, R. Balasubramonian, A. Davis, and A. N. Udipi, "Quantifying the relationship between the power delivery network and architectural policies in a 3d-stacked memory device," in *MICRO*, 2013.

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

[41] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: from error visibility to structural similarity," in *IEEE transactions on image processing*, 2004.

[42] L.-Y. Wei, "Tile-based texture mapping on graphics hardware," in *SIGGRAPH*, 2004.

[43] S. J. E. Wilton and N. P. Jouppi, "Cacti: an enhanced cache access and cycle time model," in *JSSC*, May 1996.

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

[45] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *RTAS*, April 2013.

[46] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, "Top-pim: Throughput-oriented programmable processing in memory," in *HPDC*, 2014.