GPUpd: A Fast and Scalable Multi-GPU Architecture Using Cooperative Projection and Distribution

Youngsok Kim
Seoul National University
youngsok@snu.ac.kr

Jae-Eon Jo
POSTECH
jojaeeon@postech.ac.kr

Hanhwi Jang
POSTECH
hanhwi@postech.ac.kr

Minsoo Rhu
POSTECH
minsoo.rhu@postech.ac.kr

Hanjun Kim
POSTECH
hanjun@postech.ac.kr

Jangwoo Kim
Seoul National University
jangwoo@snu.ac.kr

ABSTRACT

Graphics Processing Unit (GPU) vendors have been scaling single-GPU architectures to satisfy the ever-increasing user demands for faster graphics processing. However, as it gets extremely difficult to further scale single-GPU architectures, the vendors are aiming to achieve the scaled performance by simultaneously using multiple GPUs connected with newly developed, fast inter-GPU networks (e.g., NVIDIA NVLink, AMD XDMA). With fast inter-GPU networks, it is now promising to employ split frame rendering (SFR) which improves both frame rate and single-frame latency by assigning disjoint regions of a frame to different GPUs. Unfortunately, the scalability of current SFR implementations is seriously limited as they suffer from a large amount of redundant computation among GPUs.

This paper proposes *GPUpd*, a novel multi-GPU architecture for fast and scalable SFR. With small hardware extensions, GPUpd introduces a new graphics pipeline stage called *Cooperative Projection & Distribution (C-PD)* where all GPUs cooperatively project 3D objects to 2D screen and efficiently redistribute the objects to their corresponding GPUs. C-PD not only eliminates the redundant computation among GPUs, but also incurs minimal inter-GPU network traffic by transferring object IDs instead of mid-pipeline outcomes between GPUs. To further reduce the redistribution overheads, GPUpd minimizes inter-GPU synchronizations by implementing batching and runahead-execution of draw commands. Our detailed cycle-level simulations with 8 real-world game traces show that GPUpd achieves a geomean speedup of 4.98× in single-frame latency with 16 GPUs, whereas the current SFR implementations achieve only 3.07× geomean speedup which saturates on 4 or more GPUs.

CCS CONCEPTS

• **Computing methodologies** → **Graphics processors**; • **Computer systems organization** → *Distributed architectures*;

KEYWORDS

Graphics Processing Units (GPUs), Multi-GPU Systems, Split Frame Rendering (SFR), Graphics Pipeline

ACM Reference format:
Youngsok Kim, Jae-Eon Jo, Hanhwi Jang, Minsoo Rhu, Hanjun Kim, and Jangwoo Kim. 2017. GPUpd: A Fast and Scalable Multi-GPU Architecture Using Cooperative Projection and Distribution. In *Proceedings of MICRO-50, Cambridge, MA, USA, October 14–18, 2017,* 13 pages.
https://doi.org/10.1145/3123939.3123968

MICRO-50, October 14–18, 2017, Cambridge, MA, USA

ACM ISBN 978-1-4503-4952-9/17/10…$15.00
https://doi.org/10.1145/3123939.3123968

1 INTRODUCTION

Graphics Processing Units (GPUs) have been developed to accelerate graphics processing, the process of generating a 2D image from 3D models [40]. Although many recent studies discuss general-purpose computing on GPUs, GPUs should basically satisfy the high graphics processing performance demands of graphics applications. Graphics processing has been and will continue to be the dominant source of revenue for GPU vendors; NVIDIA's Q4 2016 revenue from gaming (GeForce series) and professional visualization (Quadro series) markets is 10.4 times larger than that of datacenter market and is steadily increasing [48].

As it gets extremely difficult to further scale single-GPU architectures, multi-GPU architectures are now essential to deliver sustainable and scalable performance to end users. Historically, GPU vendors have been scaling single-GPU architectures using transistor scaling to satisfy the high performance demands of graphics applications. However, as Moore's law slows down and Dennard scaling comes to an end [26], the number of transistors architects can put inside a given die area cannot increase at historical rates, limiting the performance scalability of a single GPU [16]. In the meantime, the need for high-performance graphics processing is ever increasing (e.g., 4K UHD games, virtual reality, augmented reality), making it crucial to provide sustainable and scalable performance to the end customers of a GPU using it for graphics processing purposes. Given that state-of-the-art single GPUs do not provide sufficiently high performance for 4K gaming [38] and virtual reality is 7 times more performance demanding than 1080p gaming [49], the need for multi-GPU graphics processing is higher than ever.

GPU vendors have been implementing alternate frame rendering (AFR) and split frame rendering (SFR) to simultaneously use multiple GPUs [1, 6, 44]. AFR improves frame rate by assigning consecutive frames to different GPUs and incurs inter-GPU synchronizations only for inter-frame data dependencies. However, newly

developed, fast inter-GPU networks (e.g., NVIDIA NVLink [47], AMD XDMA [8]) make it promising to implement SFR which assigns disjoint regions of a frame to different GPUs. By making GPUs process the same frame, SFR improves both frame rate and single-frame latency. Although SFR may incur more frequent inter-GPU synchronizations due to intra-frame data dependencies, the fast inter-GPU networks provide sufficiently low latency and high bandwidth to mitigate the performance overhead of inter-GPU synchronizations. Given the greater advantages of SFR over AFR, we focus on SFR in this work.

Despite the importance of graphics processing, the architectural design space for SFR has not been thoroughly explored. Therefore, current SFR implementations are not fast and scalable enough to provide robust performance improvements even with the fast inter-GPU networks. To achieve fast and scalable SFR, an implementation should rapidly project 3D objects to 2D screen for each GPU to identify a set of objects to process. Unfortunately, the existing SFR implementations suffer from the poor computational throughput of CPUs and a large amount of redundant computation among GPUs. First, CPU-assisted implementations [24, 32, 33] perform object projection on low-throughput CPUs, incurring significant latency as object projection is a throughput-oriented task. Second, GPU-based implementations avoid the use of CPUs by making each GPU project all objects to screen space and identify the objects it needs for its frame region [44]. However, their scalability is seriously limited as each GPU projects all the objects of a frame independently, incurring a large amount of redundant computation. Therefore, there is a need for a new SFR implementation which fully exploits the high computational throughput of GPUs without redundant work.

This paper proposes a novel multi-GPU architecture for fast and scalable SFR, called GPUpd. With small hardware extensions to graphics pipeline architecture, GPUpd introduces *Cooperative Projection & Distribution* (C-PD), a new pipeline stage at the beginning of the entire graphics pipeline. C-PD first evenly distributes all the objects to GPUs, cooperatively projects the objects to the screen space, and identifies the minimal set of objects that each GPU requires to render its disjoint region of a frame. Then, C-PD redistributes the corresponding objects to execute the rest of the graphics pipeline only with the corresponding objects. By transferring object IDs rather than large mid-pipeline outcomes, GPUpd incurs minimal additional inter-GPU network traffic. To further reduce the redistribution overheads, GPUpd also supports inter-GPU communication optimization schemes such as batching and run-ahead execution of draw commands.

Our detailed cycle-level simulations show that for 8 real-world game traces, GPUpd achieves a geomean speedup of 4.98x in single-frame latency on a 16-GPU system and the speedup can be higher with more GPUs. This demonstrates that GPUpd makes SFR fast and scalable because the current GPU-based SFR implementations achieve only 3.07x geomean speedup on a 16-GPU system, and the speedup is saturated on 4 or more GPUs. Our simulations also reveal that GPUpd is oblivious to how GPUs are interconnected with each other; GPUpd achieves similar speedups with both line and hub networks which resemble NVIDIA NVLink and AMD XDMA, respectively.

In summary, the contributions of this paper are as follows:

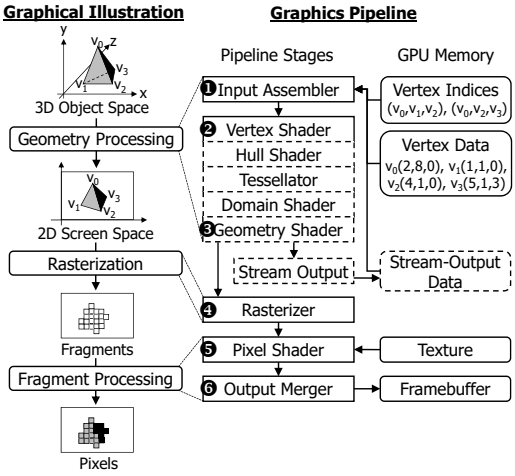


Figure 1: Graphics pipeline of DirectX 11 [41]

- A novel multi-GPU architecture called GPUpd which achieves fast, sustainable, and scalable SFR.
- A novel SFR pipeline stage called Cooperative Projection & Distribution (C-PD) which efficiently identifies the set of primitives for each GPU using high computational throughput of multiple GPUs.
- Inter-GPU communication optimizations such as batching and run-ahead execution of draw commands which reduce and hide redistribution overheads of C-PD.
- In-depth analysis of GPUpd which compares GPUpd against the current GPU-based SFR implementations in detail using cycle-level simulations.

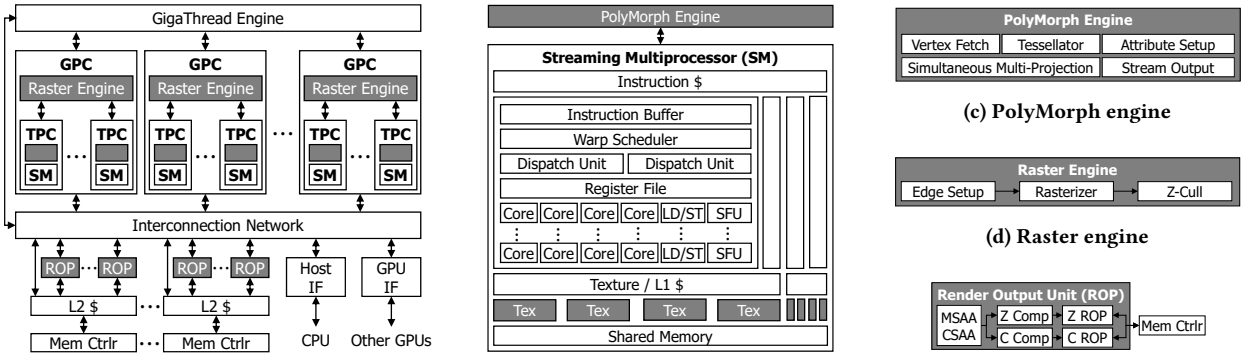
2 BACKGROUND & MOTIVATION

2.1 Graphics Pipeline

Fig. 1 shows the graphics pipeline as defined by DirectX 11 graphics application programming interface (API) [41]. Although we describe DirectX 11's graphics pipeline, it is similar to those of other recent graphics APIs (e.g., OpenGL [53]). According to the pipeline, a GPU generates pixels from application-issued 3D objects (e.g., points, lines, triangles) and writes the pixels to a framebuffer (FB), a GPU memory region storing the pixels of a frame. After processing all 3D objects of the frame, the pixels stored in the FB get displayed on display devices attached to the GPU (e.g., monitors).

The pipeline stages can be grouped into three categories.

Geometry Processing: Geometry processing step projects vertices in a 3D object space to the 2D screen space and groups the vertices into *primitives*, the basic elements GPUs operate on (typically triangles). The step consists of Input Assembler (IA), Vertex Shader (VS), Hull Shader (HS), Tessellator (Tess), Domain Shader (DS), and Geometry Shader (GS). First, IA reads application-issued vertex indices and attributes (e.g., color, depth) from GPU memory. Then, VS calculates the normalized device coordinates of each vertex, and projects the vertices on a 2D virtual coordinate system whose lower-left and upper-right corners correspond to (0, 0) and (1, 1). After VS, the projected vertices get grouped into primitives as specified by the application. At this point, the application



(a) Full-chip block diagram (b) Texture Processing Cluster (TPC) (c) PolyMorph engine (d) Raster engine (e) Render output unit (ROP) [51]

Figure 2: NVIDIA GP104 microarchitecture [49]. The components dedicated to graphics processing are shaded.

may enable HS, Tess, and DS for tessellation, a process of subdividing low-detail primitives into higher-detail ones, and/or GS to create/discard primitives. Then, the primitives get projected onto 2D screen space with fixed-function units and get transferred to the rest of the graphics pipeline.

Rasterization & Fragment Processing: Rasterization and fragment processing generate pixels from the primitives processed by geometry processing. First, Rasterizer stage converts primitives into fragments which will be used to produce pixels. Second, Pixel Shader (PS, also known as Fragment Shader) stage calculates per-fragment attributes such as color and depth. Third, Output Merger (OM) stage determines if a fragment is necessary to produce a frame, and generates pixels from the necessary fragments. For instance, depth test (Z test) finds invisible fragments which are overlapped by the current pixel and alpha test checks if the color of a pixel is within the application-specified range. As the last step, the graphics pipeline updates FB using the generated pixels.

2.2 Baseline GPU Microarchitecture

Fig. 2 shows NVIDIA GP104 microarchitecture of recent NVIDIA Pascal GPUs [49]. It consists of Graphics Processing Clusters (GPCs), Render Output Units (ROPs), L2 caches, memory controllers, a GigaThread Engine, and interfaces to CPU and other GPUs. GPCs perform computational tasks (e.g., shader execution). Each GPC consists of a Raster Engine (RE, Fig. 2d), a hardware unit for rasterization, and Texture Processor Clusters (TPCs, Fig. 2b). Each TPC contains a PolyMorph Engine (PME, Fig. 2c) and a Streaming Multiprocessor (SM) consisting of computational resources such as warp schedulers, arithmetic logic units (Cores), texture units (Texs), and shared memory. PMEs implement the fixed-function operations of the graphics pipeline other than rasterization, and SMs execute programmable shaders. GPU memory contents such as textures and FBs are interleaved across memory partitions to increase memory bandwidth. Each memory controller is paired to an L2 cache, and multiple ROPs are associated to each L2 cache. ROPs validate pixels generated by GPCs (via depth and alpha tests) and update the FB with valid pixels (Fig. 2e). For the purpose, each ROP consists of hardware units for anti-aliasing, pixel compression, and FB updates. The remaining components connect intra-GPU components and external devices. Data transfers between GPCs, ROPs, and L2 caches go through the interconnection network. Using the

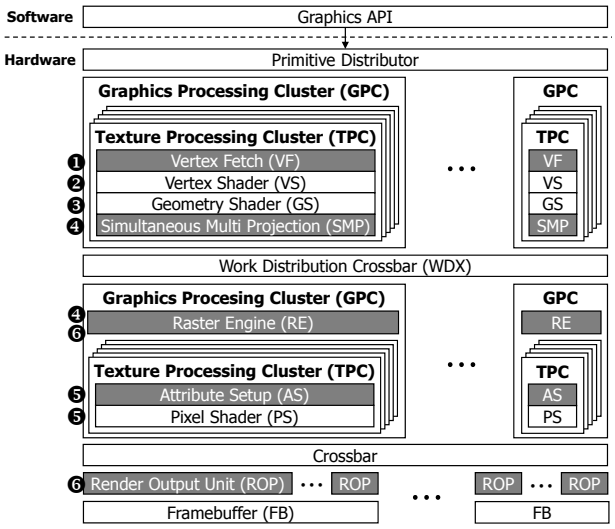


Figure 3: Execution flow of the graphics pipeline with an optional GS on NVIDIA GP104 microarchitecture (inferred from [36]). Numbers next to hardware components denote corresponding graphics pipeline stages in Fig. 1.

Host Interface (Host IF), the GPU can retrieve commands (e.g., draw commands) from CPU cores and send/receive data to/from CPU memory. Inter-GPU data transfers utilize the GPU Interface (GPU IF) connected to an inter-GPU network.

Fig. 3 shows how the graphics pipeline is executed on top of the baseline GPU microarchitecture when an optional GS is enabled. When an application issues a draw command, the GigaThread Engine receives application-issued primitives and distributes them across GPCs. Then, each TPC fetches vertex data using the Vertex Fetch (VF) unit, executes VS and GS on SMs, and calculates per-primitive screen-space coordinates using the Simultaneous Multi Projection (SMP) unit. The processed primitives are then distributed across GPCs and get rasterized into fragments by per-GPC REs. After that, a TPC setups per-primitive attributes using the Attribute Setup (AS) unit, executes PS on SMs, and distributes fragments across ROPs according to their screen-space coordinates. Lastly, each ROP validates the fragments and updates the FB using valid fragments.

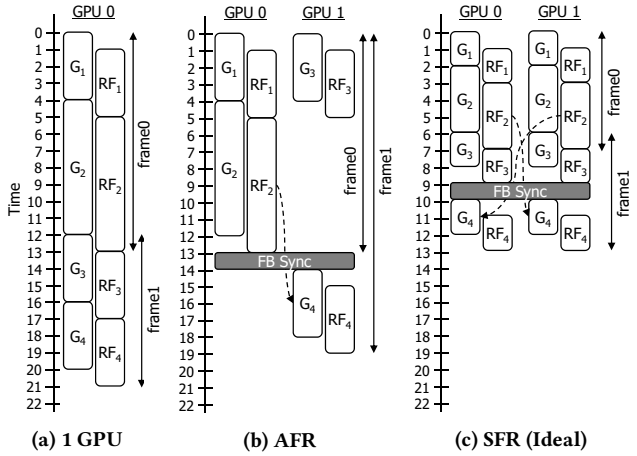


Figure 4: Working models of multi-GPU mechanisms (G: geometry processing, RF: rasterization and fragment processing, FB Sync: FB synchronization). Dashed lines denote data dependencies between two draw commands.

2.3 Multi-GPU Mechanisms

Alternate Frame Rendering (AFR). To increase frame rates, AFR distributes multiple consecutive frames to different GPUs in a temporal manner. Fig. 4b illustrates how AFR processes two consecutive frames using two GPUs. AFR assigns one frame (i.e., draw commands 1 and 2) to one GPU and the other frame (i.e., draw commands 3 and 4) to the other. If there is an inter-frame data dependency such as the dependency from draw command 2 to draw command 4 in the example, AFR synchronizes FBs via an inter-GPU network.

Although AFR can increase frame rates, it does not reduce single-frame latency as a frame is still processed by only one GPU (i.e., frame 0 in the figure). Moreover, AFR can incur higher latency when data dependencies exist between frames (i.e., frame 1 in the figure), which is common in recent game engines. In summary, AFR may be effective for processing continuous frames, but does not fit well for interactive applications demanding low single-frame latencies.

Split Frame Rendering (SFR). SFR splits a frame in a spatial manner to reduce single-frame latency by making all GPUs process disjoint regions of the same frame. Since each GPU processes only the primitives which fall into its frame region, SFR reduces the amount of work per GPU, decreasing single-frame latency (Fig. 4c). Utilizing multiple GPUs to process the same frame incurs inter-GPU synchronizations for both inter- and intra-frame data dependencies; however, the high performance of newly-developed, fast inter-GPU networks such as NVIDIA NVLink [47] and AMD XDMA [8] can mitigate inter-GPU synchronization overheads, making SFR an implementation requirement of recent graphics APIs (e.g., DirectX 12 [3]). Ideally, SFR should linearly decrease single-frame latency as the number of GPUs increases because the amount of work per GPU is typically proportional to the size of per-GPU disjoint region [25].

2.4 Limitations of Existing Implementations

SFR aims to reduce single-frame latency by making GPUs cooperatively process the same frame. To do so, an SFR implementation must identify the list of primitives necessary to a GPU to correctly process its frame region. Unfortunately, it is infeasible to statically identify per-GPU primitive lists; the screen-space coordinates of a primitive are unknown without projecting it onto 2D screen space. Thus, geometry processing must be performed for the identification.

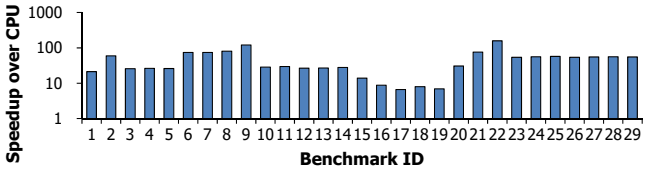


Figure 5: Speedup in geometry processing latency of NVIDIA Titan X GPU over Intel Core i7-2600 CPU with glmark2 [4]

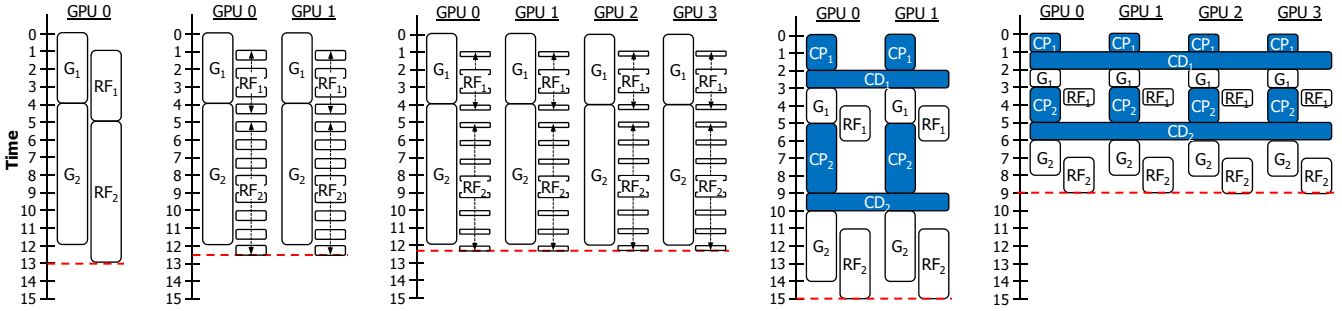
Depending on how geometry processing is performed, existing SFR implementations can be classified into two categories: CPU-assisted and GPU-based implementations.

CPU-assisted Implementations. CPU-assisted implementations [24, 32, 33] execute geometry processing on CPU cores. After projecting primitives onto 2D screen space using CPU cores, the implementations distribute a primitive to the GPUs demanding the primitive. Then, each GPU executes the graphics pipeline on the retrieved primitives to process its frame region. The implementations use CPUs instead of GPUs as they assume traditional low-performance inter-GPU networks such as NVIDIA SLI and AMD CrossFireX bridges.

Unfortunately, the implementations greatly suffer from the low computational throughput of CPU cores as geometry processing is a throughput-oriented task. To demonstrate the limitation, we compare the geometry processing latency of glmark2 [4] benchmarks on Intel Core i7-2600 CPU (quad-core, 3.80 GHz) and NVIDIA Titan X GPU. Both the CPU and the GPU are configured to perform only geometry processing using `GL_RASTERIZER_DISCARD` flag; rasterization stage discards all primitives. Profiling results show that the GPU significantly outperforms the CPU, achieving the maximum speedup of 156.7x over the CPU (Fig. 5). Thus, avoiding the use of CPU cores for geometry processing is essential to achieve fast and scalable SFR.

GPU-based Implementations. To avoid the use of CPU cores, GPU vendors implement GPU-based SFR by making each GPU perform geometry processing on all primitives of a frame, and then process only the primitives necessary to its frame region. By using high-throughput GPUs, the implementations significantly outperform CPU-assisted ones. In addition, no additional hardware modification is necessary to existing single-GPU architectures as they already implement geometry processing as part of the graphics pipeline.

However, the existing GPU-based implementations are still neither fast enough nor scalable. First, as each GPU must perform geometry processing on all primitives of a frame, geometry processing latency remains intact regardless of the number of GPUs; only the amount of rasterization and fragment processing gets reduced due to smaller per-GPU frame regions. Moreover, as rasterization and fragment processing depend on the output of geometry processing, the overall performance improvement gets bounded by geometry processing whose latency remains intact. Fig. 6b illustrates such a limitation. Although the current SFR reduces `RF1` and



(a) 1 GPU (b) Current (2 GPUs) (c) Current (4 GPUs) (d) C-PD (2 GPUs) (e) C-PD (4 GPUs)

Figure 6: Working models of the current GPU-based implementations and C-PD for the first frame in Fig. 4 (G: geometry processing, RF: rasterization and fragment processing, CP: cooperative projection, CD: cooperative distribution)

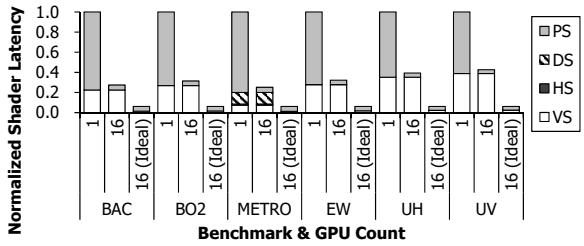


Figure 7: Breakdown of the shader latency on a single NVIDIA Titan X GPU and the projected shader latencies of current and ideal 16-GPU SFR implementations[1]. GS is excluded as no benchmark utilizes it.

RF2 by half, the current SFR keeps G1 and G2 intact. Due to a dependence from geometry processing to rasterization and fragment processing, RF1 and RF2 are stalled waiting for G1 and G2. What is worse is that most of the results from G1 and G2 are not necessary. A recent characterization study on 3D game traces reveals that up to 72% of application-issued primitives are discarded before rasterization stage due to clipping and culling [52]. More primitives get discarded on multi-GPU SFR as per-GPU frame region shrinks.

Second, the current implementations are not scalable as geometry processing is performed redundantly across all GPUs. Comparing Fig. 6b and Fig. 6c illustrates the limited scalability. Though the number of GPUs increases from 2 to 4, the latency is not reduced as the critical path on geometry processing is not changed; just RF becomes about half less busy than before. The redundant geometry processing seriously limits the scalability as geometry processing latency would dominate the overall latency with higher GPU counts (Fig. 7). Thus, to achieve fast and fully-scalable SFR, it is crucial to reduce not only the latency of rasterization and fragment processing, but also geometry processing latency as the number of GPUs increases.

3 COOPERATIVE PROJECTION & DISTRIBUTION

For fast and scalable SFR, we propose a new graphics pipeline stage, *Cooperative Projection & Distribution* (C-PD), before geometry processing stage of the original graphics pipeline. C-PD stage identifies the primitives necessary to each GPU before a GPU processes its frame region (Fig. 6d, Fig. 6e). C-PD consists of two phases: cooperative projection (CP) and cooperative distribution (CD) phases. As the first step of CP phase, C-PD evenly distributes application-issued primitives to GPUs. Each GPU performs geometry processing on its primitives and generates their screen-space coordinates. With the screen-space coordinates of a primitive, C-PD identifies which GPUs demand the primitive for their frame regions and constructs a list of target GPUs for the primitive. Then, CD phase begins by making each GPU transmit its primitive IDs to the target GPUs. At the same time, each GPU receives the primitive IDs for its frame region. After GPUs finish exchanging primitive IDs, each GPU executes the original graphics pipeline using the new set of primitive IDs and processes its target frame region.

C-PD has several advantages over the current GPU-based SFR architectures. First, C-PD accelerates geometry processing by minimizing the number of primitives each GPU processes. Especially, while the current architectures require all GPUs to perform geometry processing for all primitives of a frame, C-PD achieves the same goal by performing geometry processing only *once* on each primitive. Moreover, since each primitive is independent from others, CP phase can scale on multiple GPUs; geometry processing latency linearly decreases as the number of GPUs increases (Fig 6e).

Despite of the advantages, implementing C-PD can be challenging as C-PD involves a number of inter-GPU interactions to exchange primitive IDs between GPUs. Thus, a careful design which optimizes the inter-GPU interactions is necessary to minimize the negative performance impacts of high inter-GPU communication latency.

4 GPUPD ARCHITECTURE

This section presents GPUpd, a novel and scalable multi-GPU architecture implementing C-PD. Fig. 8 shows how GPUpd executes the extended graphics pipeline.

4.1 Software Extensions

When an application issues a draw command (e.g., `DrawPrimitive()`), GPUpd begins CP phase by evenly distributing primitives to GPUs.

[1] Batman: Arkham City (BAC), Call of Duty: Black Ops II (BO2), Metro: Last Light Redux (METRO), The Evil Within (EW), Unigine Heaven (UH), Unigine Valley (UV)

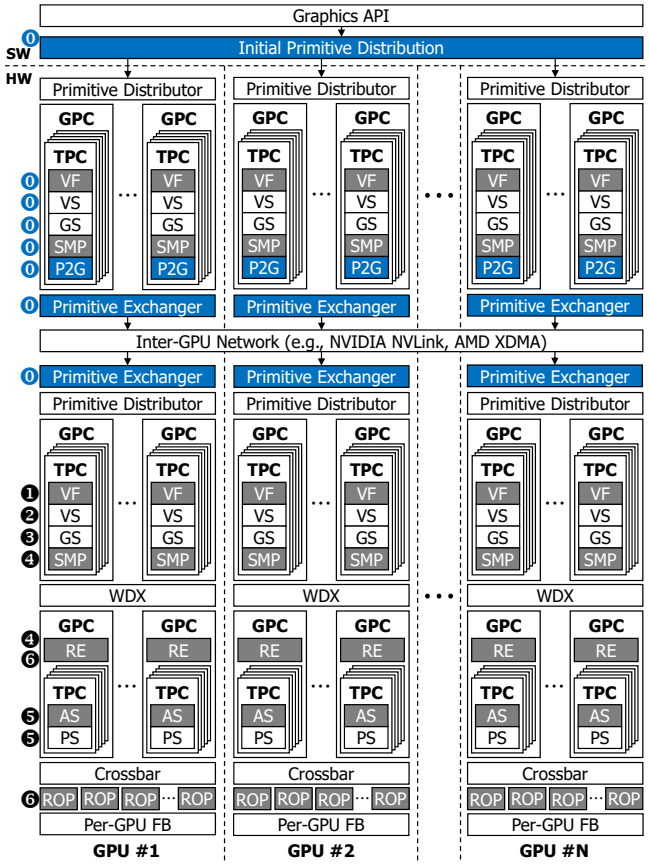


Figure 8: Execution flow of the graphics pipeline on GPUpd. Numbers next to hardware components denote corresponding graphics pipeline stages in Fig. 1, and the 0-th stage denotes C-PD.

Upon receiving a draw command whose primitive count is m and the first primitive ID is fid, the extended graphics API runtime reinterprets the draw command as a set of n draw commands on an n-GPU system; the i-th draw command is set to have a primitive count of $\lceil \frac{m}{n} \rceil$ and a first primitive ID of $fid + \lceil \frac{m}{n} \rceil \times i$. After that, the runtime issues the i-th draw command to the i-th GPU.

Note that the runtime groups consecutive primitives into the same draw command. This is because our baseline GPU architecture is an immediate-mode rendering (IMR) architecture which processes primitives in the order issued by the application. Thus, unless the runtime does not employ such a grouping scheme, GPUpd must reorder the primitives back to the application-specified order during execution. By grouping consecutive primitives, GPUpd avoids the cost of implementing the reordering logic.

4.2 Hardware Extensions

Projecting Primitives Using Existing Hardware. After distributing primitives across GPUs using the software extensions, GPUpd begins the CP phase of the draw command by making GPUs project their primitives onto 2D screen space. To minimize implementation costs, GPUpd uses the existing hardware units for primitive projection. For instance, a GPU projects its primitives by distributing

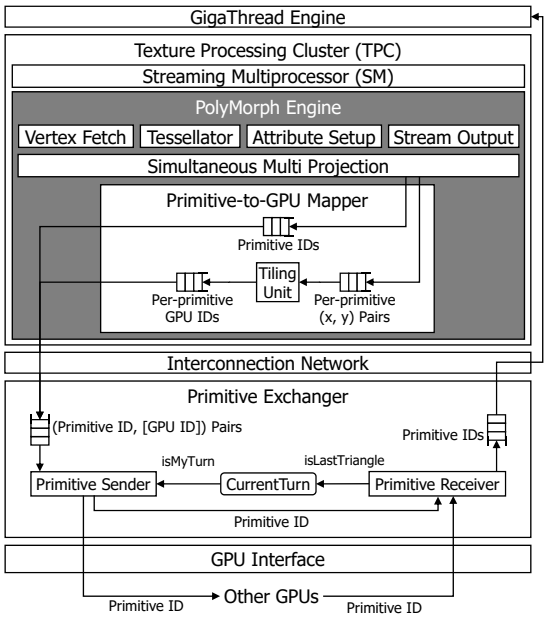
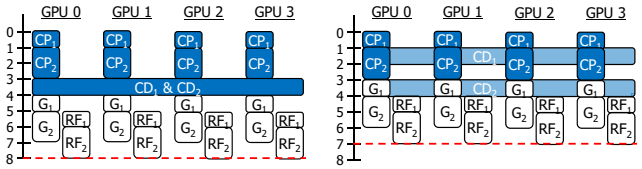


Figure 9: Hardware extensions for GPUpd. Per-PME Primitive-to-GPU Mapper units (for CP phases) and a Primitive Exchanger unit (for CD phases) are newly added.

them across TPCs, executing vertex and geometry shaders on SMs, and using per-PME vertex fetch and simultaneous multi projection (SMP) units (Fig. 8). As the last stage of geometry processing, SMP units produce the screen-space coordinates along with other attributes (e.g., per-vertex colors) of a primitive.

Primitive-to-GPU Mapper Units. After projecting its set of primitives onto 2D screen space, a GPU needs to identify the list of GPUs demanding a primitive to correctly process its frame region. To do so, a new hardware unit is necessary as the baseline GPU microarchitecture lacks relevant hardware units; it only determines whether a primitive is visible or not via early depth/Z tests. For the purpose, GPUpd adds a *primitive-to-GPU mapper (P2G)* unit to each PME as shown in Fig. 9. Motivated by the fact that GPUs split a frame in a spatial manner with SFR, P2G units utilize the screen-space coordinates of a primitive to identify the list of GPUs demanding the primitive as follows. First, a P2G unit retrieves a primitive's screen-space coordinates and other attributes from the associated SMP unit. Second, the unit queues only the ID and the screen-space coordinates of the primitive as other attributes are unnecessary for primitive-to-GPU mapping. Third, the unit calculates the minimal bounding box on 2D screen space which can fully contain the primitive. By testing whether the screen-space area the bounding box lies on overlaps with per-GPU frame regions, the unit generates an n-bit bitmask indicating which of the n GPUs demands the primitive; if the i-th bit is set, the i-th GPU demands the primitive. The generated n-bit bitmasks are then used to exchange primitive IDs between GPUs.

Primitive Exchanger Units. Using the n-bit bitmasks generated by a P2G unit, the CD phase of a draw command may start by making a GPU distribute its primitive IDs to other GPUs. However, naïvely transmitting primitive IDs demands additional on-chip



(a) Batching (b) Runahead

Figure 10: Working models of inter-GPU communication optimizations for the first frame in Fig. 4

storage and hardware units to guarantee functional correctness. This is because the baseline GPU architecture is an IMR architecture; primitives must be processed in the application-specified order. Thus, to guarantee functional correctness, a GPU must have both sufficiently large on-chip storage to buffer the primitive IDs from other GPUs and a hardware unit which sorts retrieved primitive IDs into the application-specified order. Note that a draw command may issue a large number of primitives, typically up to 4,294,967,295 (i.e., the maximum value of `unsigned int` data type).

To avoid large implementation costs, GPUpd adds a *Primitive Exchanger (PE)* unit to each GPU. PE units exchange primitive IDs between them by exploiting the way the extended API runtime distributes primitives to GPUs (Sec. 4.1) to eliminate the need for large on-chip storage. Using PE units, GPUpd implements CD phases using a turn-based mechanism as follows. Initially, the PE unit of the first GPU (i.e., GPU 0) distributes its primitives according to the bitmasks. After distributing all of its primitives, the first GPU broadcasts a last-triangle marker to denote that it has finished transmitting its primitives. Then, the second GPU (i.e., GPU 1) begins distributing its primitives, and broadcasts a last-triangle marker. This procedure continues until all GPUs finish distributing their primitives. In this way, GPUpd ensures that the order of primitives received by a GPU is the same as specified by the application.

After the PE unit of a GPU completes exchanging primitive IDs with other GPUs, the GPU begins to process its target frame region by executing the original graphics pipeline with the new list of primitive IDs from the PE unit.

4.3 Inter-GPU Communication Optimizations

Draw Command Batching. A naïve GPUpd implementation would perform C-PD for every draw command; however, doing so can significantly increase single-frame latency as each GPU must receive last-triangle markers from all other GPUs for every draw command. To reduce inter-GPU synchronizations, GPUpd implements *draw command batching* by further extending the software-side runtime (Fig. 10a); it merges a number of consecutive draw commands into a single, larger draw command. By doing so, fewer number of CD phases occur as the number of draw commands decreases, improving latency. When merging multiple draw commands, the extended API runtime guarantees functional correctness by not merging consecutive draw commands having read-after-write data dependencies.

Although merging draw commands can be beneficial, merging as many draw commands as possible does not necessarily guarantee lower single-frame latency. This is because load imbalance between GPUs can occur as draw commands have different geometry processing latencies, and the runtime issues consecutive primitives of a draw command to a GPU (Sec. 4.1). For instance, if a merged draw command consists of many primitives with low geometry processing latency followed by a few primitives with high geometry processing latency, GPUs having smaller IDs finish geometry processing much earlier than the other GPUs do.

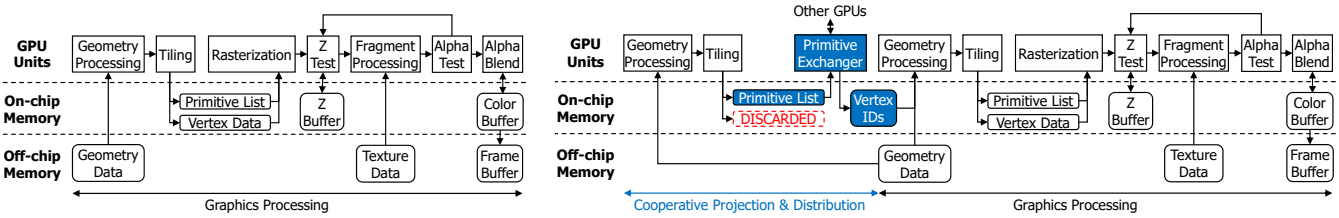
To prevent such load imbalance, draw command batching employs *batching threshold*. Batching threshold prevents overly aggressive merging of draw commands; when merging consecutive draw commands, the runtime stops merging more if the currently merged primitive count exceeds batching threshold. After issuing the currently merged draw commands as a single draw command to GPUs, draw command batching works on a new merge starting with the next draw command. GPUpd uses 4096 primitives per draw command as it provides the highest geomean speedup. We later perform a sensitivity study on batching threshold.

Runahead Execution. Employing draw command batching helps GPUpd reduce the number of CD phases (and thus inter-GPU synchronizations); however, the computational resources of a GPU can still become idle for long periods of time as the PE unit of the GPU must retrieve last-triangle markers from all other GPUs to complete a CD phase. To further improve single-frame latency by utilizing the idle resources, GPUpd implements *runahead execution* which allows the CP phase of the next draw command to begin executing alongside the CD phase of the current draw command (Fig. 10b). By doing so, runahead execution can not only utilize the idle resources of a GPU, but also improve single-frame latency by overlapping the execution of multiple draw commands.

Implementing runahead execution demands PE units to buffer not only primitive IDs, but also their draw command IDs to guarantee functional correctness; as the baseline GPU architecture implements IMR, GPUpd should prevent the primitive IDs of the next draw command from being distributed earlier than those of the current draw command. However, we find that the overhead is low as only 11 bits are necessary to encode draw command IDs for our benchmarks; the maximum number of draw commands does not exceed 1840. If a larger number of draw commands needs to be supported, architects can freely increase the number of bits for encoding draw command IDs.

4.4 Tile-based Rendering Architectures

Up to this point, we have assumed that the baseline GPU architecture employs IMR which executes the entire graphics pipeline in one pass for each of application-issued primitives. On the other hand, *tile-based rendering (TBR)* architectures employ a two-step execution model with on-chip Z buffer and FB to reduce off-chip memory bandwidth consumption (Fig. 11a). After performing geometry processing on application-issued primitives, TBR architectures bin the processed primitives into small rectangular regions, called tiles, using their Tiling units. When all the primitives are tiled, they perform rasterization and fragment processing on a per-tile basis, updating the on-chip buffers rather than off-chip memory. By accessing on-chip buffers instead of off-chip memory, TBR architectures significantly improve energy efficiency. Many GPU architectures designed for high energy efficiency employ TBR (e.g., Tiled Caching of NVIDIA Maxwell/Pascal [42], ARM Mali [30], Qualcomm Adreno [7], Imagination PowerVR [57]).



(a) Graphics pipeline on TBR architectures

(b) Example implementation of C-PD on top of TBR architectures

Figure 11: Implementation of C-PD on top of tile-based rendering (TBR) architectures

Per-GPU Resources	
Clock Frequency	1 GHz
Unified Shader	32 shaders, 16 simd4+scalar ALUs / shader
ROP	64 ROPs, 4 fragments / cycle / ROP
GDDR Memory	10 channels, 8 banks / channel, 2 KB rows Interleaving Granularity: 256 B / channel
Split Frame Rendering	
Interleaving Granularity	32-by-32 square pixels / GPU
Inter-GPU Synchronization	Broadcast updated Z buffer and FB contents upon a read-after-write dependency
Inter-GPU Network	
Topology	line (similar to NVIDIA NVLink [47]), hub (similar to AMD XDMA [8])
Line	GPU-GPU Link Latency: 100, 200, 400, 800 ns GPU-GPU Link Bandwidth: 40 GB/s
Hub	GPU-Hub Link Latency: 100, 200, 400, 800 ns GPU-Hub Link Bandwidth: 40 GB/s Hub Bandwidth: 80 GB/s

Table 1: Simulator configuration

Name	Abbr.	Resolution	Draw Count	Inter-GPU Sync.	
				Count	Size [MB]
Call of Duty 2	cod2	640x480	1005	27	22.92
Crysis	cry	800x600	1067	25	73.51
GRID	grid	1280x1024	1840	19	143.47
Mirror's Edge	mirrors	1280x1024	1257	16	114.93
Need For Speed: Undercover	nfs	1280x1024	1262	13	77.49
S.T.A.L.K.E.R.: Call of Pripyat	stal	1280x1024	1086	7	98.05
Unreal Tournament 3	ut3	1280x1024	876	12	80.99
Wolfenstein	wolf	640x480	1697	13	20.25

Table 2: Description of the evaluated game traces

(a) cod2 (b) cry (c) grid (d) mirrors

(e) nfs (f) stal (g) ut3 (h) wolf

Figure 12: Processing results of the game traces

Since C-PD is a pre-processing pipeline stage, it can be easily implemented on top of TBR architectures (Fig. 11b). The example implementation operates as follows. First, it begins a CP phase by performing geometry processing on the primitives assigned by the software-side runtime. Second, Tiling unit maps each projected primitive to the corresponding GPUs and stores the mapping information to on-chip buffer. At this point, the vertex data (i.e., the outcome of geometry processing) are discarded as C-PD does not utilize them. Third, the implementation performs a CD phase using the primitive list stored in on-chip memory. The vertex IDs, derived from the primitives IDs from other GPUs, get stored in the free on-chip memory space. Then, the implementation begins the original graphics pipeline stages using the vertex IDs stored in on-chip memory and the geometry data stored in off-chip memory.

Implementation costs become lower with TBR architectures as they already contain hardware units useful for C-PD. First, as TBR architectures are already equipped with Tiling unit which maps a projected primitive to the corresponding tiles, C-PD can utilize it to map a projected primitive to the corresponding GPUs. Second, during CP phases, we can utilize the on-chip memory region dedicated to the vertex data to store the vertex IDs to process. This is possible as C-PD buffers only primitives' screen-space coordinates and vertex IDs, which are small enough to fit in on-chip memory, while discarding the other outcomes of geometry processing.

5 EVALUATION

This work evaluates GPUpd by extending ATTILA [22, 46] to model the microarchitectural changes of GPUpd. While we acknowledge that ATTILA models fairly outdated AMD R600 microarchitecture [9], we configure its parameters to resemble those of NVIDIA GP104 microarchitecture (Table 1). Our simulation framework implements SFR by interleaving 32-by-32 square pixels of a frame to different GPUs. We validated our GPU-based SFR architecture model with real-world benchmarking results [17, 56]; our two-GPU SFR model reduces single-frame latency by 60.4% on geomean which is on par with the benchmarking results.

We evaluate GPUpd with single-frame traces from eight DirectX-based, real-world games (Table 2 and Fig. 12). As SFR demands inter-GPU buffer synchronizations to correctly process a frame, our simulation framework invokes an inter-GPU buffer synchronization upon a read-after-write data dependency on depth-stencil and render-target buffer contents. We extend ATTILA's graphics API runtime to track depth-stencil and render-target buffer reads (e.g., `PSSetShaderResources()` in DirectX 11) and writes (e.g., `OMSetRenderTargets()` in DirectX 11). Then, upon a data-dependent draw command (e.g., `DrawIndexed()` in DirectX 11), the runtime makes GPUs broadcast the updated contents of their depth-stencil and render-target buffers to other GPUs in turn. The number and the amount of inter-GPU buffer synchronizations are shown in 'Inter-GPU Sync.' column of Table 3.

We model a high-bandwidth inter-GPU network using parameters similar to those of NVIDIA NVLink [47], a point-to-point (P2P) network supporting up to 4 GPUs where each GPU has 4 ports and each port provides 20 GB/s of bandwidth. We model NVLink as a line network rather than a P2P network for better scalability with

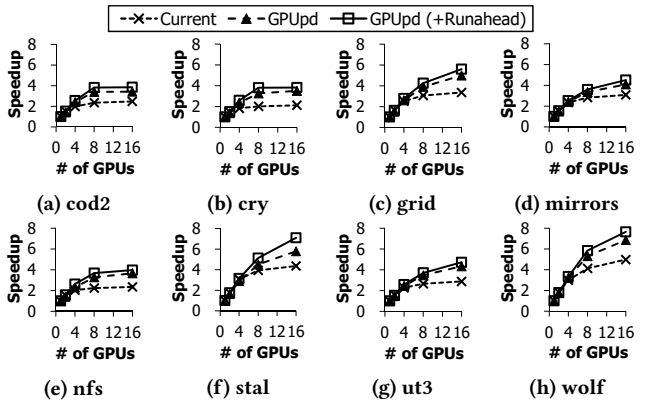


Figure 13: Speedups in single-frame latency of the current GPU-based SFR architecture and GPUpd. 'Current' denotes the current SFR architecture, and 'GPUpd (+Runahead)' denotes GPUpd with runahead execution.

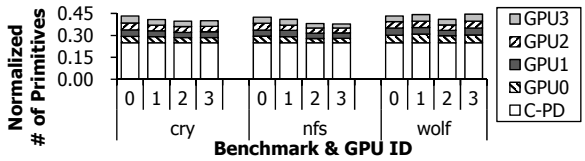


Figure 14: Distribution of the sources of the primitives processed by each GPU on 4-GPU C-PD

4+ GPUs. 400 ns is the default latency of each link; 100 ns, 200 ns, and 800 ns are also used for a sensitivity analysis. The latency is added whenever a packet flows through a link (e.g., at least 800 ns to transfer data from GPU 0 to GPU 2).

5.1 Scalable Graphics Performance

Fig. 13 shows the speedups in single-frame latency of the current GPU-based SFR architecture, GPUpd, and GPUpd with runahead execution over single-GPU processing. Up to 16 GPUs are configured to evaluate scalability. Draw command batching is applied to GPUpd and GPUpd with runahead execution. The batching threshold is set to 4096 as the geomean speedup saturates at 4096.

The results show that GPUpd delivers scalable performance improvements as the number of GPUs increases while the current architecture suffers from the saturated speedup with 4+ GPUs. GPUpd with runahead execution achieves the maximum speedup of 7.65x and a geomean speedup of 4.98x; GPUpd without runahead execution achieves the maximum speedup of 6.86x and a geomean speedup of 4.45x. On the other hand, the current architecture achieves a geomean speedup of 3.07x due to redundant geometry processing. In summary, GPUpd provides scalable performance improvements by preventing a primitive from being projected more than once, and effectively hiding inter-GPU communication latency with two communication optimizations.

GPUpd achieves scalable performance with multiple GPUs by making GPUs cooperatively project objects and then process only the necessary objects. Fig. 14 shows that each GPU projects only a quarter of a frame's objects during C-PD stage followed by the execution of the original graphics pipeline to produce a frame on

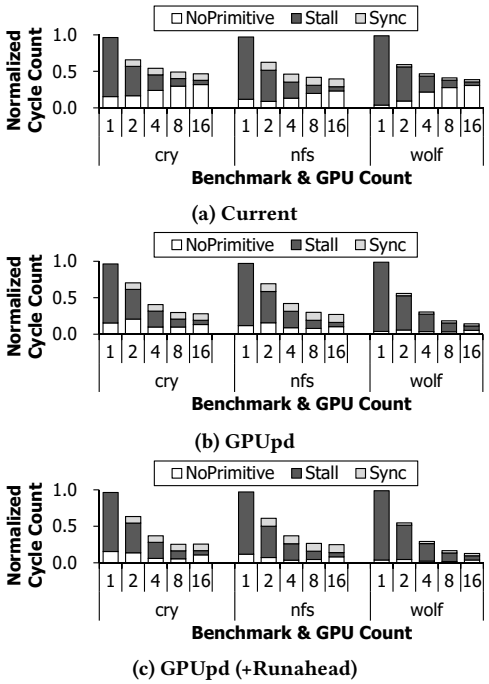


Figure 15: Cycle stacks constructed with the activities at the boundary of geometry processing and rasterization

a 4-GPU setup. By doing so, GPUpd can reduce the number of primitives processed per GPU by up to 62% when compared to the current architecture.

5.2 In-depth Performance Analysis

In this experiment, we perform a detailed analysis of the varying performance improvements of GPUpd. Three traces are selected for the analysis: `cry`, `nfs`, and `wolf`. GPUpd with runahead execution achieves the maximum speedup of 7.65x with `wolf`, and the minimum speedup of 3.82x with `cry` with 16 GPUs. `nfs` is chosen as runahead execution provides the minimum additional speedup gain of 0.31x.

To analyze why GPUpd achieves different speedups for the benchmarks, we construct a cycle stack using the per-cycle activities at the boundary of geometry processing and rasterization stages. The cycle stack consists of three categories: NoPrimitive, Stall, and Sync. First, NoPrimitive denotes the cycles when no primitive is available for either rasterization or CD phases. Second, a cycle is counted as Stall if an available primitive cannot be issued to rasterization stage due to pipeline back pressure. Third, Sync is the number of cycles spent for inter-GPU synchronizations due to inter-draw dependencies (e.g., render-to-texture). Note that the cycles spent to issue primitives to either rasterization stage or PE units are omitted; such cycles account for less than 1%.

Fig. 15 shows the cycle stacks of the current architecture and GPUpd. First, both the current architecture and GPUpd reduce the number of Stall cycles as the number of GPUs increases. This is because each GPU processes only a small portion of a frame, incurring less rasterization and fragment processing work. Second, the number of NoPrimitive cycles of the current architecture steadily increases with larger GPU counts as primitive projection overheads

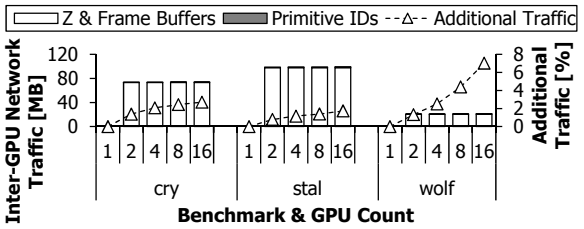


Figure 16: Breakdown of inter-GPU network traffic

remain intact. However, GPUpd effectively reduces the number of NoPrimitive cycles by scaling down geometry processing overheads with C-PD. Third, GPUpd with runahead execution further reduces the number of NoPrimitive cycles by executing the C-PD of next draw commands to utilize idle hardware units.

GPUpd achieves the largest speedup with `wolf` by effectively reducing the number of NoPrimitive cycles. The minimum speedup with `cry` is due to the fact that `cry`'s primitives are necessary to multiple GPUs; such primitives increase rasterization and fragment processing latencies, limiting the reduction in NoPrimitive cycles. Runahead execution does not reduce NoPrimitive cycles much with `nfs`, reducing the impact of runahead execution. Despite of achieving varying speedups, GPUpd always outperforms the current architecture by scaling down geometry processing overheads and by hiding inter-GPU communication overheads.

5.3 Inter-GPU Network Overheads

GPUpd incurs additional inter-GPU network traffic as GPUs distribute their primitives over the network. To minimize the additional network traffic, GPUpd exchanges primitive IDs rather than mid-pipeline outcomes which can be as large as a few hundreds of kilobytes. Fig. 16 shows the additional network traffic GPUpd incurs in addition to the FB synchronization overheads of the current architecture. Even with 16 GPUs, GPUpd incurs less than 7.03% of additional network traffic by using primitive IDs. Note that GPUpd incurs larger network traffic with `wolf` as its FB synchronization overhead is much smaller than those of `cry` and `nfs` (Table 2). By transferring small primitive IDs rather than mid-pipeline outcomes, GPUpd effectively minimizes the additional network traffic.

5.4 Sensitivity Analyses

Inter-GPU Network Latency. As the key idea of C-PD is to distribute projected primitives via inter-GPU communication, effectively hiding inter-GPU network latency is essential for fast and scalable SFR. To minimize the exposure of the network latency, GPUpd implements two optimizations: draw command batching and runahead execution. In this experiment, we evaluate their effectiveness by measuring the speedups of GPUpd while changing inter-GPU communication latency. We use 4096 as batching threshold and evaluate the three benchmarks of interest (chosen in Sec. 5.2). Note that GPUpd is less sensitive to network bandwidth as the size of primitive IDs is small.

Fig. 17 shows the speedup in single-frame latency with varying inter-GPU communication latencies on 4- and 16-GPU systems. We observe that GPUpd sustains the maximum speedup even if inter-GPU network latency grows up to 800 ns. 16-GPU GPUpd

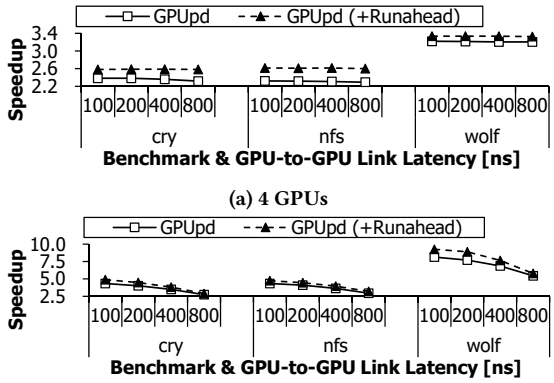


Figure 17: Sensitivity to inter-GPU network latency

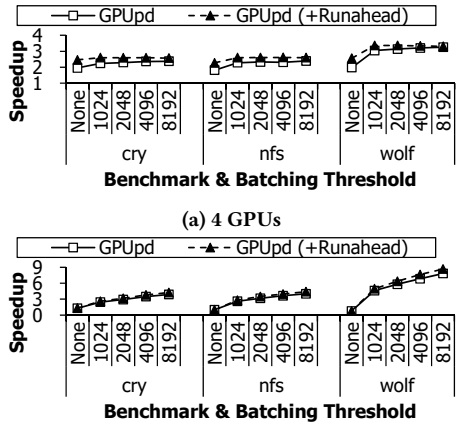


Figure 18: Sensitivity to batching threshold. 'None' denotes that none of draw commands are batched.

is more sensitive to the network latency as each GPU needs to retrieve last-triangle markers from 15 GPUs rather than 3 GPUs; however, the reduced speedups are still higher than the current SFR architecture. In summary, the two optimizations of GPUpd effectively hide inter-GPU communication latencies.

Batching Threshold. In this experiment, we perform a sensitivity analysis on batching threshold of draw command batching. Draw command batching improves performance by reducing the number of inter-GPU synchronizations. However, careless selection of batching threshold can introduce load imbalance between GPUs as geometry processing latency may differ between draw commands.

Fig. 18 shows the speedup of GPUpd as batching threshold varies. Draw command batching is especially effective for the 16-GPU case; disabling draw command batching results in higher single-frame latency than single-GPU processing. The results from 4-GPU GPUpd confirm that blindly increasing batching threshold does not necessarily deliver higher speedups due to load imbalance. On 4-GPU GPUpd with runahead execution, the speedup saturates at 2048 and begins to decrease as batching threshold increases; however, the difference is not significant and GPUpd still outperforms the current SFR architecture.

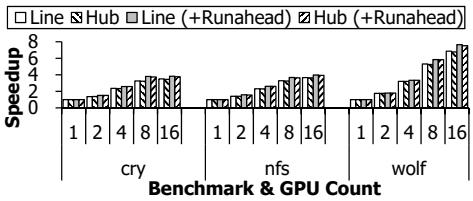


Figure 19: Sensitivity to inter-GPU network topology

Name	Abbr.	Resolution	Draw Count	Inter-GPU Sync.	
				Count	Size [MB]
UNIGINE Heaven	heaven	1920x1080	1519	46	233.73
UNIGINE Valley	valley	1920x1080	1026	41	142.22

Table 3: Description of the profiled DirectX 11 traces

(a) heaven **(b) valley**

Figure 20: Processing results of the profiled DirectX 11 traces

Inter-GPU Network Topology. Our default inter-GPU network is an NVLink-like line network; however, some GPU vendors utilize PCI Express (PCIe) to implement fast inter-GPU networks (e.g., AMD XDMA [8]). To evaluate how GPUpd performs with such PCIe-based networks, we model hub network consisting of a central switch and multiple switch-GPU links. The central switch forwards packets sent from GPUs to their destination GPUs, and is configured to provide a throughput of 40 GB/s. Each switch-GPU link is configured to provide 40-GB/s bandwidth and 400-ns latency.

Fig. 19 compares the speedups of GPUpd on line and hub networks. For all of the three benchmarks of interest, the line network offers slightly higher speedups as it provides lower minimum GPU-to-GPU communication latency (400 ns of line vs. 800 ns of hub); however, the performance difference is very small (less than 1.82%). Despite of having longer minimum communication latency, the average communication latency of the hub network is lower than the line network, helping the hub network mitigate the longer minimum communication latency. This is because farther GPUs on the line network goes through multiple GPU-to-GPU links and each GPU-to-GPU communication incurs 400 ns of latency. In summary, GPUpd provides higher speedups on both line and hub networks, making it effective even with PCIe-like networks.

5.5 Recent Graphics Stacks

As ATTILA models outdated AMD R600 microarchitecture and supports outdated DirectX 9 applications, the experimental results from ATTILA may not reflect the performance characteristics of state-of-the-art microarchitectures (e.g., NVIDIA GP104) and graphics APIs (e.g., DirectX 11). To evaluate whether GPUpd is effective with recent graphics stacks, we profile DirectX 11 gaming benchmarks on recent NVIDIA GTX 970 GPU and analyze how much performance gains GPUpd can potentially provide.

For the analysis, we profile single-frame, geometry processing, and inter-GPU buffer synchronization latencies of the two frames

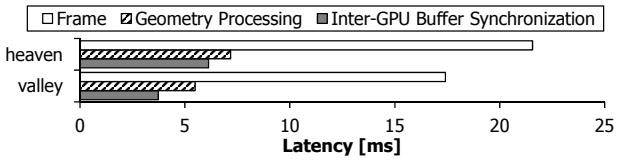


Figure 21: Single-frame, geometry processing, and inter-GPU buffer synchronization latencies of the profiled traces

from representative DirectX 11-based, 1080p benchmarks (Table 3 and Fig. 20). We select the three latencies as geometry processing latency should account for much of single-frame latency for GPUpd to be effective. In addition, inter-GPU buffer synchronization latency should be low enough to make SFR effective.

To measure the latencies, we utilize NVIDIA Nsight [5] and application trace analysis. Single-frame latency and geometry processing of the two traces are measured with Nsight. To calculate inter-GPU buffer synchronization latency, we first collect the DirectX 11 API traces of the two frames using apitrace [2]. Then, we feed the API traces to a custom Python script which identifies read-after-write dependencies between API function calls. The Python script tracks data dependencies similar to how the extended ATTILA's graphics API runtime does. After that, we calculate inter-GPU buffer synchronization latency by assuming a 40-GB/s inter-GPU network (similar to NVLink and PCIe Gen3 x16 links) to calculate inter-GPU buffer synchronization latency.

Fig. 21 shows the profiled latencies of the two frames. The measurement shows that the performance gains would be less than 38.3% with current SFR architectures due to redundant geometry processing; the combined latencies of geometry processing and inter-GPU buffer synchronization account for 61.7% and 52.9% of the single-frame latency for heaven and valley, respectively. On the other hand, GPUpd can reduce single-frame latency by up to 66.7% (for valley) and 68.5% (for heaven) as it reduces geometry processing latency with C-PD. Hence, GPUpd has large potential performance gains even with recent hardware and graphics APIs.

6 RELATED WORK

Parallel & Multi-GPU Graphics Processing. Depending on when the data for the frame region of a GPU get determined during graphics processing, SFR implementations are often classified into three categories: sort-first, sort-middle, and sort-last architectures [43].

First, sort-first architecture calculates per-GPU primitive sets before executing the graphics pipeline by projecting primitives to 2D screen space beforehand (e.g., WireGL [31, 32], Chromium [33], Equalizer [24]). Although each GPU would process minimal number of primitives, existing implementations using CPU cores greatly suffer from the low computational throughput of CPU cores.

Second, sort-middle architecture makes GPUs exchange primitive attributes between geometry processing and rasterization stages. For instance, on Pixel-Planes 5 [28], rasterizer stage distributes its outcomes to multiple ROPs. Other example implementations include NVIDIA's CUDA-based software rasterizer [37] and Intel's Larrabee [54]. As the size of geometry processing outcomes is very large (hundreds of kilobytes per primitive), sort-middle architecture demands much higher communication bandwidth than current inter-GPU networks can provide.

Third, sort-last architecture makes GPUs exchange fragments after fragment processing (e.g., PixelFlow [27]). Each GPU then assembles the retrieved fragments to generate its frame region. Similar to sort-middle architecture, sort-last architecture demands high network bandwidth for fragment distribution [43], incurring high communication latency.

Fourth, sort-everywhere architecture such as RealityEngine [11] and Pomegranate [25] aims for more fine-grained load balancing by distributing work before geometry processing, between geometry processing and rasterization, between rasterization and fragment processing, and after fragment processing. Targeted to balance workload at every pipeline stage, the architecture incurs a significantly large amount of communication traffic, making it not suitable for multi-GPU systems with limited network bandwidth.

Exploiting Mid-pipeline Outcomes. Recent work by de Lucas et al. [21] exploits mid-pipeline outcomes to accelerate collision detection on mobile devices. GPUpd also utilizes mid-pipeline outcomes; however, GPUpd exploits geometry processing outcomes, not those of rasterization, and targets fast and scalable SFR. In addition, GPUpd is orthogonal to the work as C-PD occurs before the graphics pipeline.

Architectural Improvements for Graphics Processing. To improve single-frame latency and frame rate, researchers and GPU vendors have proposed various architecture improvements for graphics processing. Recent GPUs aim to improve the utilization of computational hardware units by employing unified shaders [39, 45] and scalar ALUs [10, 18] rather than discrete geometry/fragment shaders and VLIW ALUs, respectively. As unified shaders can execute any programmable shaders and scalar ALUs do not demand vectorization, GPUs can achieve higher utilization for applications with different shader loads and non-vectorizable code.

Some studies propose architectural support to minimize the off-chip memory bandwidth consumed by texture data, a dominant off-chip memory bandwidth consumer on GPUs [23]. Texture caches [20, 29] are widely employed by GPUs as they can effectively reduce the consumption. As follow-up work, prefetching for texture caches has been widely explored [12, 34, 35, 50, 55, 58]. Decoupling texture accesses from fragment shaders [13] and exploiting texture re-use between consecutive frames [14] have been explored as well. Xie et al. [59] explore the use of processing-in-memory to reduce texture memory traffic.

Other studies focus on improving the energy efficiency of graphics processing. To achieve the goal, Arnau et al. [15] reduce redundant fragment shader executions by caching the shaded results, and AMFS [19] re-uses fragment shader outcomes for nearby samples.

7 CONCLUSION

GPUpd is a fast and scalable multi-GPU architecture for SFR which implements C-PD, a new graphics pipeline stage to identify and redistribute primitive IDs each GPU demands for its frame region. GPUpd optimizes inter-GPU communication by batching and run-ahead executing draw commands to minimize the redistribution overheads. For real-world DirectX-based game traces, GPUpd achieves a geomean speedup of 4.98x in single-frame latency on a 16-GPU system, whereas the current architecture achieves only a geomean speedup of 3.07x which saturates on 4 or more GPUs.

ACKNOWLEDGMENTS

This work was supported by Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-TB1503-05.

REFERENCES

[1] [n. d.]. AMD CrossFire guide for Direct3D® 11 applications. ([n. d.]). https://gpuopen-librariesandsdks.github.io/doc/AMD-CrossFire-guide-for-Direct3D11-applications.pdf
[2] [n. d.]. apitrace. ([n. d.]). http://apitrace.github.io/
[3] [n. d.]. Direct3D 12 Graphics. ([n. d.]). https://msdn.microsoft.com/en-us/library/windows/desktop/dn903821(v=vs.85).aspx
[4] [n. d.]. glmark2. ([n. d.]). https://github.com/glmark2/glmark2
[5] [n. d.]. NVIDIA Nsight. ([n. d.]). http://www.nvidia.com/object/nsight.html
[6] 2011. SLI Best Practices. (2011). http://developer.download.nvidia.com/whitepapers/2011/SLI_Best_Practices_2011_Feb.pdf
[7] 2015. Adreno Hardware Tutorial 3: Tile Based Rendering. (2015). https://www.youtube.com/watch?v=SeySx0TkluE
[8] 2015. Modernizing multi-GPU gaming with XDMA. (2015). https://community.amd.com/community/gaming/blog/2015/05/11/modernizing-multi-gpu-gaming-with-xdma
[9] Advanced Micro Devices, Inc. 2008. R600-Family Instruction Set Architecture. (2008). http://developer.amd.com/wordpress/media/2012/10/r600isa.pdf
[10] Advanced Micro Devices, Inc. 2012. AMD's Graphics Core Next Technology. (2012). https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf
[11] Kurt Akeley. 1993. RealityEngine Graphics. In *Proc. 20th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*.
[12] Bruce Anderson, Andy Stewart, Rob MacAulay, and Turner Whitted. 1997. Accommodating Memory Latency In A Low-cost Rasterizer. In *Proc. ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware (HWWS)*.
[13] José-María Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. 2012. Boosting Mobile GPU Performance with a Decoupled Access/Execute Fragment Processor. In *Proc. 39th IEEE/ACM International Symposium on Computer Architecture (ISCA)*.
[14] Jose-Maria Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. 2013. Parallel Frame Rendering: Trading Responsiveness for Energy on a Mobile GPU. In *Proc. 22nd IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)*.
[15] Jose-Maria Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. 2014. Eliminating Redundant Fragment Shader Executions on a Mobile GPU via Hardware Memoization. In *Proc. 41st IEEE/ACM International Symposium on Computer Architecture (ISCA)*.
[16] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, and David Nellans. 2017. MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability. In *Proc. 44th International Symposium on Computer Architecture (ISCA)*.
[17] Steve Burke. 2016. GTX 1060 "SLI" Benchmark - Outperforms GTX 1080 with Explicit Multi-GPU. (2016). http://www.gamersnexus.net/guides/2519-gtx-1060-sli-benchmark-in-ashes-multi-gpu
[18] Hessed Choi. 2016. Bifrost - The GPU architecture for next five billion. (2016). https://www.arm.com/files/pdf/20160628_A04_ATF_Korea_Hessed_Choi.pdf
[19] Petrik Clarberg, Robert Toth, Jon Hasselgren, Jim Nilsson, and Tomas Akenine-Möller. 2014. AMFS: Adaptive Multi-Frequency Shading for Future Graphics Processors. In *Proc. 41st International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*.
[20] Michael Cox, Narendra Bhandari, and Michael Shantz. 1998. Multi-Level Texture Caching for 3D Graphics Hardware. In *Proc. 25th Annual International Symposium on Computer Architecture (ISCA)*.
[21] Enrique de Lucas, Pedro Marcuello, Joan-Manuel Parcerisa, and Antonio González. 2015. Ultra-Low Power Render-Based Collision Detection for CPU/GPU Systems. In *Proc. 48th International Symposium on Microarchitecture (MICRO)*.
[22] Victor Moya del Barrio, Carlos González, Jordi Roca, Agustín Fernández, and Roger Espasa. 2006. ATTILA: A Cycle-Level Execution-Driven Simulator for Modern GPU Architectures. In *Proc. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.
[23] Michael Doggett. 2012. Texture Caches. *IEEE Micro* 32 (2012).
[24] Stefan Eilemann, Maxim Makhinya, and Renato Pajarola. 2009. Equalizer: A Scalable Parallel Rendering Framework. *IEEE Transactions on Visualization and Computer Graphics* 15 (2009). Issue 3.
[25] Matthew Eldridge, Homan Igehy, and Pat Hanrahan. 2000. Pomegranate: A Fully Scalable Graphics Architecture. In *Proc. 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*.
[26] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark Silicon and the End of Multicore Scaling. In *Proc. 38th International Symposium on Computer Architecture (ISCA)*.

[27] John Eyles, Steven Molnar, John Poulton, Trey Greer, Anselmo Lastra, Nick England, and Lee Westover. 1997. PixelFlow: The Realization. In *Proc. ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware (GH)*.

[28] Henry Fuchs, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, and Laura Israel. 1989. Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories. In *Proc. 16th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*.

[29] Ziyad S. Hakura and Anoop Gupta. 1997. The Design and Analysis of a Cache Architecture for Texture Mapping. In *Proc. 24th Annual International Symposium on Computer Architecture (ISCA)*.

[30] Peter Harris. 2014. The Mali GPU: An Abstract Machine, Part 2 - Tile-based Rendering. (2014). https://community.arm.com/graphics/b/blog/posts/the-mali-gpu-an-abstract-machine-part-2---tile-based-rendering

[31] Greg Humphreys, Ian Buck, Matthew Eldridge, and Pat Hanrahan. 2000. Distributed Rendering for Scalable Displays. In *Proc. ACM/IEEE Conference on Supercomputing (SC)*.

[32] Greg Humphreys, Matthew Eldridge, Ian Buck, Gordon Stoll, Matthew Everett, and Pat Hanrahan. 2001. WireGL: A Scalable Graphics System for Clusters. In *Proc. 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*.

[33] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. 2002. Chromium: A Stream-Processing Framework for Interactive Rendering on Clusters. In *Proc. 29th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*.

[34] Homan Igehy, Matthew Eldridge, and Kekoa Proudfoot. 1998. Prefetching in a Texture Cache Architecture. In *Proc. ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware (HWWS)*.

[35] Mark J. Kilgard. 1997. Realizing OpenGL: Two Implementations of One Architecture. In *Proc. ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware (HWWS)*.

[36] Christoph Kubisch. 2015. Life of a triangle - NVIDIA's logical pipeline. (2015). https://developer.nvidia.com/content/life-triangle-nvidias-logical-pipeline

[37] Samuli Laine and Tero Karras. 2011. High-Performance Software Rasterization on GPUs. In *Proc. ACM SIGGRAPH Symposium on High Performance Graphics (HPG)*.

[38] Jeremy Laird. 2016. NVIDIA GTX 1080: A Big Leap, But Not Quite A 4K Slayer. (2016). https://www.rockpapershotgun.com/2016/07/14/gtx-1080-4k-performance/

[39] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. 2008. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro* 28 (2008). Issue 2.

[40] David Luebke and Greg Humphreys. 2007. How GPUs Work. *IEEE Computer* 40 (2007). Issue 2.

[41] Microsoft Corporation. [n. d.]. Graphics Pipeline. ([n. d.]). https://msdn.microsoft.com/en-us/library/windows/desktop/ff476882(v=vs.85).aspx

[42] David Mitchelson. 2017. NVIDIA GTX 1080 Ti Review. (2017). https://www.vortez.net/articles_pages/nvidia_gtx_1080_ti_review,2.html

[43] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. 1994. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applications* 14, 4 (1994).

[44] Jordi Roca Monfort and Mark Grossman. 2009. Scaling of 3D Game Engine Workloads on Modern Multi-GPU Systems. In *Proc. Conference on High Performance Graphics (HPG)*.

[45] Victor Moya, Carlos González, Jordi Roca, Agustín Fernández, and Roger Espasa. 2005. A Single (Unified) Shader GPU Microarchitecture for Embedded Systems. In *Proc. 1st International Conference on High Performance Embedded Architectures and Compilers (HiPEAC)*.

[46] Victor Moya, Carlos Gonzalez, Jordi Roca, Agustin Fernandez, and Roger Espasa. 2005. Shader Performance Analysis on a Modern GPU Architecture. In *Proc. 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[47] NVIDIA Corporation. 2014. NVIDIA® NVLink™ High-Speed Interconnect: Application Performance. (2014). http://info.nvidianews.com/rs/nvidia/images/NVIDIA%20NVLink%20High-Speed%20Interconnect%20Application%20Performance%20Brief.pdf

[48] NVIDIA Corporation. 2016. NVIDIA Announces Financial Results for the Fourth Quarter and Fiscal 2016. (2016). http://nvidianews.nvidia.com/news/nvidia-announces-financial-results-for-the-fourth-quarter-and-fiscal-2016

[49] NVIDIA Corporation. 2016. NVIDIA GeForce GTX 1080. (2016). http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf

[50] Martin Randall. 1997. Talisman: Multimedia for the PC. *IEEE Micro* 17 (1997). Issue 2.

[51] Ashu Rege. [n. d.]. An Introduction to Modern GPU Architecture. ([n. d.]). http://download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf

[52] Jordi Roca, Victor Moya, Carlos González, Chema Solís, and Agustín Fernández. 2006. Workload Characterization of 3D Games. In *Proc. IEEE International Symposium on Workload Characterization (IISWC)*.

[53] Mark Segal and Kurt Akeley. 2017. The OpenGL® Graphics System: A Specification (Version 4.5 (Core Profile) - June 29, 2017). (2017). https://khronos.org/registry/OpenGL/specs/gl/glspec45.core.pdf

[54] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. 2008. Larrabee: A Many-Core x86 Architecture for Visual Computing. In *Proc. 35th International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*.

[55] B. V. N. Silpa, Anjul Patney, Tushar Krishna, Preeti Ranjan Panda, and G. S. Visweswaran. 2008. Texture Filter Memory – a power-efficient and scalable texture memory architecture for mobile graphics processors. In *Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

[56] Ryan Smith. 2013. The AMD Radeon R9 290X Review. (2013). http://www.anandtech.com/show/7457/the-radeon-r9-290x-review

[57] Rys Sommefeldt. 2015. A look at the PowerVR graphics architecture: Tile-based rendering. (2015). https://www.imgtec.com/blog/a-look-at-the-powervr-graphics-architecture-tile-based-rendering/

[58] Jay Torborg and James T. Kajiya. 1996. Talisman: Commodity Realtime 3D Graphics for the PC. In *Proc. 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*.

[59] Chenhao Xie, Shuaiwen Leon Song, and Jing Wang. 2017. Processing-in-Memory Enabled Graphics Processors for 3D Rendering. In *Proc. 23rd IEEE Symposium on High Performance Computer Architecture (HPCA)*.