CHOPIN: Scalable Graphics Rendering in Multi-GPU Systems via Parallel Image Composition

Xiaowei Ren Mieszko Lis
The University of British Columbia
{xiaowei, mieszko}@ece.ubc.ca

***Abstract*—The appetite for higher and higher 3D graphics quality continues to drive GPU computing requirements. To satisfy these demands, GPU vendors are moving towards new architectures, such as MCM-GPU and multi-GPUs, that connect multiple chip modules or GPUs with high-speed links (e.g., NVLink and XGMI) to provide higher computing capability.**

Unfortunately, it is not clear how to adequately parallelize the rendering pipeline to take advantage of these resources while maintaining low rendering latencies. Current implementations of Split Frame Rendering (SFR) are bottlenecked by redundant computations and sequential inter-GPU synchronization, and fail to scale as the GPU count increases.

In this paper, we propose CHOPIN, a novel SFR scheme for multi-GPU systems that exploits the parallelism available in image composition to eliminate the bottlenecks inherent to existing solutions. CHOPIN composes opaque sub-images out-of-order, and leverages the associativity of image composition to compose adjacent sub-images of transparent objects asynchronously. To mitigate load imbalance across GPUs and avoid inter-GPU network congestion, CHOPIN includes two new scheduling mechanisms: a draw-command scheduler and an image composition scheduler. Detailed cycle-level simulations on eight real-world game traces show that, in an 8-GPU system, CHOPIN offers speedups of up to 1.56× (1.25× gmean) compared to the best prior SFR implementation.

I. Introduction

Graphics Processing Units (GPUs) were originally developed to accelerate graphics processing — the process of generating 2D images from 3D models [30]. Although much recent computer architecture research has focused on using GPUs for general-purpose computing, high-performance graphics processing has historically accounted for the lion's share of demand for GPUs. This continues to be the case, with graphics remaining the dominant source of revenue for GPU vendors: for example, NVIDIA's year 2019 revenues from the gaming (GeForce) and professional visualization (Quadro) markets combined are 2.5× and 11.5× higher than that from the datacenter and automotive markets, respectively [51]. This is driven by many applications, including gaming, scientific data visualization, computer-aided design, virtual reality (VR), augmented reality (AR), and so on. Gaming itself continues to evolve: 4K and VR gaming demand 4× and 7× more performance than 1080p HD gaming, respectively [3, 46], while modern games have millions or billions of triangles often smaller than a pixel [8].

This need for substantial performance improvements has, however, been increasingly difficult to satisfy with conventional single-chip GPU systems. To continue scaling GPU performance, GPU vendors have recently built larger systems [47, 49, 50] that rely on distributed architectures such as Multi-Chip-Module GPU (MCM-GPU) [15] and multi-GPUs [34, 56, 67]. MCM-GPU and multi-GPU systems promise to push the frontiers of performance scaling much further by connecting multiple GPU chip modules (GPMs) or GPUs with advanced packaging [55] and networking technologies, such as NVIDIA's NVLink [42], NVSwitch [43], and AMD's XGMI [1]. In principle, these platforms can offer substantial opportunities for performance improvement; in practice, however, their performance tradeoffs for graphics processing are different from that of single-chip GPUs, and fully realizing the benefits requires the use of distributed rendering algorithms.

Distributed rendering is, of course, not new: GPU vendors have long combined two to four GPUs using SLI [45] and Crossfire [11]. These distribute the rendering workload using either alternate frame rendering (AFR), where different GPUs process consecutive frames, or split-frame rendering (SFR), which assigns disjoint regions of a single frame to different GPUs. AFR processes alternate frames independently and improves the *average* frame rate, but does nothing to improve the *instantaneous* frame rate, which can be significantly lower than the average frame rate. This problem, called micro-stuttering, is inherent to AFR, and can result in a dramatically degraded gameplay experience [2, 5, 7]. In contrast, SFR can improve *both* the frame rate and the single-frame latencies [20, 27, 40]. Because of this, SFR is more widely used in practice, and we focus on SFR in this paper. The tradeoff, however, is that SFR requires GPUs to exchange data for both inter- and intra-frame data dependencies, which creates significant bandwidth and latency challenges for the inter-GPU interconnect.

While the recent introduction of high-performance interconnects like NVLink and XGMI targets the inter-GPU communication bandwidth constraints, key challenges still remain. SFR assigns split screen regions to separate GPUs, but the mapping of primitive (typically triangle) coordinates to screen regions is not known ahead of time, and must be computed before distributing work among GPUs. CPU preprocessing techniques exist [20, 26, 27], but are limited by low throughput. GPU methods rely on redundant computation, where every GPU projects *all* 3D primitives to the 2D screen space and retains only the primitives in its own screen region. Unfortunately, this does not scale to modern workloads where triangle counts have grown much faster than resolutions [8]. The recent proposal GPUpd [28] attempts to take advantage

2378-203X/21/$31.00 ©2021 IEEE
DOI 10.1109/HPCA51647.2021.00065

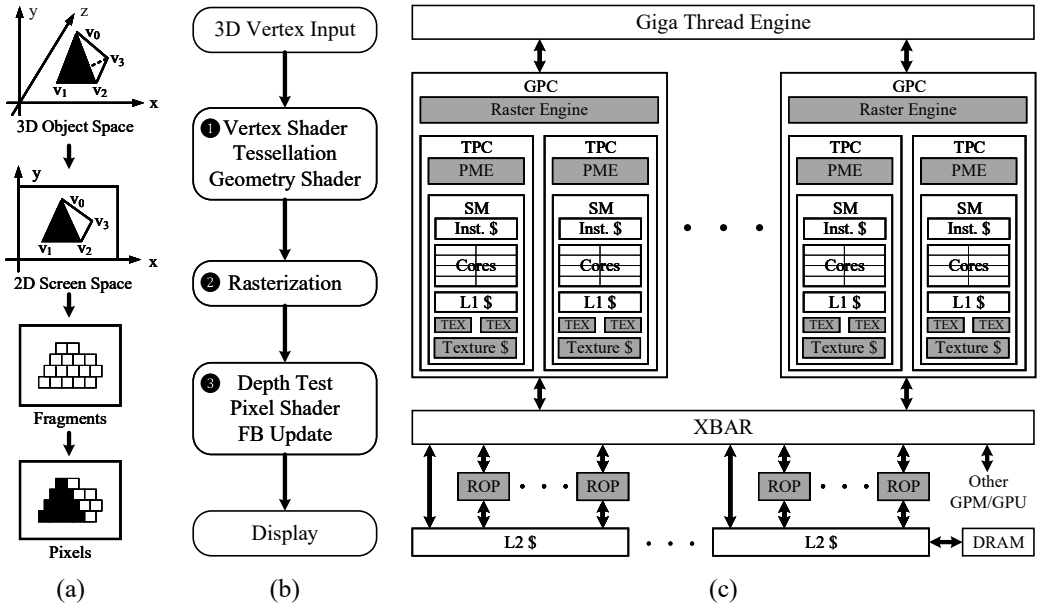


Fig. 1. Graphics processing: (a) data being processed; (b) rendering operations; (c) GPU microarchitecture (components specific to rendering are shaded).

of the new high-speed interconnects to reduce the redundant computation, but is still bottlenecked by sequential inter-GPU primitive redistribution needed to preserve the input order of primitives (see Section III-A for a detailed analysis of prior SFR solutions). Therefore, there is an urgent need for parallel rendering schemes that can leverage today's high-speed interconnects and reliably scale to multi-GPU systems.

In this paper, we propose CHOPIN, an SFR technique that eliminates the performance overheads of prior solutions by leveraging parallel image composition. Draw commands are distributed across different GPUs to remove redundant computation, and image composition is performed in parallel to obviate the need for sequential primitive exchange.

CHOPIN includes a novel draw command scheduler to balance the workload among the GPUs, and a novel image composition scheduler to reduce the network congestion that can easily result from naïve inter-GPU sub-image exchange.

Overall, this paper makes the following contributions:

- We trace the main performance cost of existing SFR mechanisms to redundant computation and sequential inter-GPU communication requirements.
- We propose CHOPIN, a parallel composition technique that takes advantage of the parallelism available in image composition to remove overheads of prior SFR solutions.
- We develop a draw command scheduler and an image composition scheduler to address load imbalance and interconnect congestion challenges in distributed rendering.

Through an in-depth analysis using cycle-level simulations on a range of real-world game traces, we demonstrate that CHOPIN outperforms the prior state-of-the-art SFR implementation by up to 1.56× (1.25× gmean) in an 8-GPU system.

II. Background

A. The 3D Rendering Pipeline

The 3D graphics pipeline, shown in Fig. 1(a), projects a 3D scene with objects that often consist of thousands of primitives (usually triangles) onto a 2D screen space. On the screen, primitives end up as thousands of pixels, which are accumulated in a framebuffer (FB) and sent to the display once rendering is complete. Producing a single frame typically involves thousands of draw commands to render all objects in the scene, all of which must go through the graphics pipeline.

Fig. 1(b) shows the graphics pipeline defined by DirectX [18]; other pipelines (e.g., OpenGL [57]) are similar. The key pipeline stages are geometry processing, rasterization, and fragment processing. Geometry processing ❶ first reads vertex attributes (e.g., 3D coordinates) from memory and projects them to 2D screen coordinates using vertex shaders. Vertices are grouped into primitives (typically triangles); some primitives may then be split into multiple triangles through tessellation, which creates smoother object surfaces for a higher level of visual detail. Generated primitives that are outside of the current viewport are then culled by geometry-related shaders. The next stage, called rasterization ❷, converts primitives into fragments, which will in turn be composed to produce pixels on the screen. Fragment processing ❸ first performs the depth test (Z test) which discards occluded fragments; the surviving fragments then have their attributes (e.g., color) computed using a pixel shader. Finally, the shaded fragments (which may be opaque or semi-transparent) are composed to generate the pixels, which are written to the framebuffer and eventually sent to the display.

B. The Graphics GPU Architecture

Fig. 1(c) illustrates the overall microarchitecture of NVIDIA's Turing GPU [44]. It consists of multiple Graphics Processing Clusters (GPCs), which are connected to multiple Rendering Output Units (ROPs) and L2 cache slices via a high-throughput crossbar. A Giga Thread Engine distributes the rendering workload to the GPCs based on resource availability.

GPCs perform rasterization in dedicated Raster Engines, and organize resources into multiple Texture Processing Clusters (TPCs). Within each TPC, the PolyMorph Engine (PME) performs most non-programmable operations of the graphics pipeline except rasterization (e.g., vertex fetching, tessellation, etc.). The Streaming Multiprocessor (SM), which comprises hundreds of shader cores, executes the programmable parts of the rendering pipeline, such as the vertex shaders, pixel shaders, etc.; to reduce hardware complexity, SMs schedule and execute threads in SIMD fashion (i.e., warps of 32 or 64 threads). The Texture Unit (TEX) is a hardware component that samples 2D textures to map them onto 3D objects.

On the other side of the interconnect, ROPs perform fragment-granularity tasks such as the depth test, anti-aliasing, pixel compression, pixel blending, and pixel output to the framebuffer. A shared L2 cache, accessed through the crossbar, buffers the data read from off-chip DRAM.

C. MCM-GPU and Multi-GPU Architectures

In the past decade, transistor density improvements have become harder to achieve, and may no longer be economically realistic [59]. To continue improving performance, recent work has focused on new architectures, such as MCM-GPU [15] at the package level and multi-GPUs [34] at the system level — platforms which connect multiple GPU chip modules (GPMs) or GPUs with high-performance links (e.g., NVLink [42], NVSwitch [43], or XGMI [1]).

Although GPMs/GPUs are independent hardware components, researchers have proposed high-level abstractions that present the system to programmers as if it were a single larger GPU. To exploit data locality, both MCM-GPU and multi-GPU systems schedule adjacent Cooperative Thread Arrays (CTAs) to the same GPM/GPU, with each memory page usually mapped to the GPM/GPU that first accessed it (i.e., first-touch). Different GPMs/GPUs can cache the data of each other to reduce inter-GPM/GPU communication [15, 16, 34, 67]. Scoped memory consistency models [24, 31] and hierarchical cache coherence protocols [56] have also been designed for solid and efficient inter-GPM/GPU synchronizations. However, all of this work has focused on general-purpose GPU applications rather than the graphics rendering task we target with CHOPIN.

D. Parallel Image Composition

Image composition is the reduction of several images into one, and is performed at pixel granularity. The reduction process is a sequence of operations, each of which has two inputs: the current pixel value p_{old} and the incoming value p_{new}. The two are combined using an application-dependent function f to produce the updated pixel $p = f(p_{old}, p_{new})$. The exact definition of f depends on the task: for example, f can select the pixel which is closer to the camera, or blend the color values of the two pixels. A common blending operation is the *over* operator [54] $p = p_{new} + (1 - \alpha_{new}) * p_{old}$, where p represents the pixel color and opacity components, and α is the pixel opacity only. Other blending operators include *addition*, *multiplication*, and so on.

For opaque pixels, f is commonly defined to compare the depth value and keep the pixel which is closer to the camera. Obviously, picking the smallest depth value from multiple pixels can be done out-of-order. However, composition of transparent or semi-transparent objects needs to blend multiple pixels, which in general must follow the depth order either front-to-back or back-to-front; for example, the visual effect of putting a drop of light-pink water above a piece of glass is different from the reversed order. For a series of pixels, therefore, the final value of f is derived from an ordered reduction of individual operations, $f = f_1 \circ f_2 \circ \cdots \circ f_n$. The ordering of f_1 through f_n matters, and in general the sequence cannot be permuted without altering the semantics of f. Fortunately, although blending operators are not commutative, they are associative: i.e., $f_1 \circ f_2 \circ f_3 \circ f_4 = (f_1 \circ f_2) \circ (f_3 \circ f_4)$ [17]. As we detail in Section III-B, CHOPIN leverages this associativity to compose transparent sub-images asynchronously.

Apart from the reduction function, how pixels are sent to the GPU where the reduction occurs also matters for performance. The simplest communication method is direct-send [25, 41]: once a GPU has finished processing its workload, it begins to distribute the image regions that belong to other GPUs, regardless of the readiness of the destination GPUs. With a large number of GPUs, this can easily congest the network with many simultaneous messages. To address this issue, binary-swap [32, 68] and Radix-k [53] first divide composition processes into multiple groups, and then compose sub-images with direct-send inside each group; to compose all sub-images, several rounds of this procedure are required. Alternately, Sepia [35] and Lightning-2 [60] rely on special hardware to accelerate image composition, but this incurs significant hardware costs.

In contrast, the approach we take in this paper maintains the simplicity of direct-send, and mitigates network congestion issues via a novel image composition scheduler: within CHOPIN, any two GPUs start composition-related transfers only when they are ready and available.

III. Motivation

A. Limits of Existing SFR Solutions

SFR splits the workload of a single frame into multiple partitions and distributes them among different GPUs. However, because the screen location of 3D objects depends on the camera view and is not known until after the primitive projection phase, individual GPUs must synchronize and exchange information *somewhere* along the rendering pipeline in order to produce the correct final image.

Based on where this synchronization happens, SFR implementations fall into three categories: sort-first, sort-middle, and sort-last [36]. Sort-first rendering identifies the destination

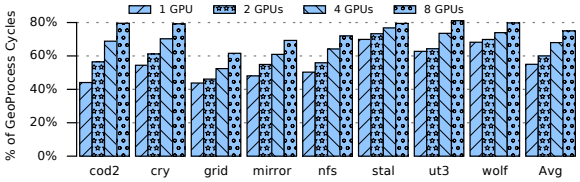


Fig. 2. Percentage of geometry processing cycles in the graphics pipeline of conventional SFR implementation. Performance is not scalable because each GPU always needs to process all primitives even though more GPUs exist.

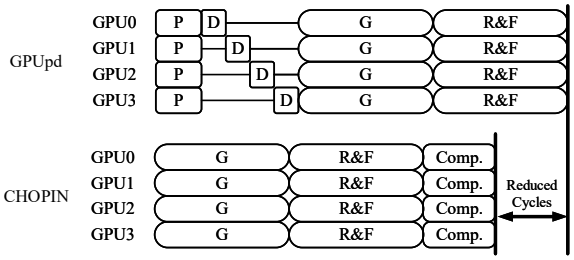


Fig. 3. Graphics pipelines of GPUpd and CHOPIN. (P: Primitive Projection, D: Primitive Distribution, G: Geometry Processing, R: Rasterization, F: Fragment Processing, Comp: Parallel Image Composition.)

GPUs of each primitive by conducting preliminary transformations at the very beginning of the graphics pipeline to compute the screen coordinates of all primitives, and distributes each primitive to the GPUs that correspond to the primitive's screen coordinates; after primitive distribution, each GPU can run the normal graphics pipeline independently. In contrast, both sort-middle and sort-last distribute primitives without knowing where they will fall in screen space, and exchange partial information later: sort-middle rendering exchanges geometry processing results *before* the rasterization stage, while sort-last rendering exchanges fragments at the end of the pipeline, before final image composition.

Among these three implementations, sort-middle is rarely adopted because the geometry processing output is very large (hundreds of KBs per primitive) [29, 58]. Both CPUs and GPUs have been used for the preliminary transformation in sort-first rendering [20, 26, 27, 38, 45]. Thanks to higher throughput, GPU-assisted implementations tend to perform better than CPUs, but they duplicate all primitives in every GPU to amortize the low bandwidth and long latency of traditional inter-GPU links [38, 45]. In these schemes, each GPU maps all primitives to screen coordinates, and eventually drops the primitives that fall outside of its assigned screen region. Unfortunately, this duplicated pre-processing stage is not scalable: as shown in Fig. 2, redundant geometry processing dominates the execution cycles of graphics pipeline and severely impacts performance as the number of GPUs grows.

To address the problem of redundant computation and take advantage of recent high-performance interconnects, Kim et al. proposed GPUpd [28], shown in Fig. 3. A sort-first technique, GPUpd attempts to evenly distribute all primitives of each draw command across the GPUs. GPUs project the received

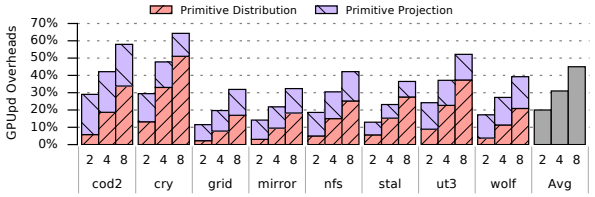


Fig. 4. Percentage of execution cycles of the extra pipeline stages in GPUpd. With more GPUs in the system, sequential primitive distribution among GPUs becomes a critical performance bottleneck.

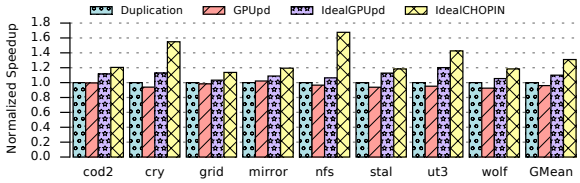


Fig. 5. Potential performance improvement afforded by leveraging parallel image composition.

primitives to screen space, and then exchange primitives based on projection results so that each GPU owns only primitives that fall into its assigned region of screen space. Finally, each GPU executes the full graphics pipeline on its primitives.

While GPUpd can reduce the overhead of redundant primitive projections, it requires GPUs to distribute primitive IDs *sequentially* to maintain the input primitive order; otherwise, a GPU would need a large memory to buffer exchanged primitive IDs and a complex sorting structure to reorder them. During the inter-GPU primitive exchange, GPU0 first distributes its primitive IDs to other GPUs, then GPU1 distributes its primitive IDs, and this procedure continues until all GPUs have completed primitive distribution. As shown in Fig. 4, with more GPUs in the system (2–8 GPUs), the sequential primitive distribution becomes a critical performance bottleneck.

B. Parallel Image Composition in CHOPIN

To eliminate the performance overhead of redundant computing and sequential inter-GPU synchronizations, in this paper, we propose CHOPIN: a *sort-last* rendering scheme with a pipeline shown in Fig. 3.

CHOPIN first divides consecutive draw commands of each frame into multiple groups based on draw command properties. Draw commands in each group are distributed across different GPUs. Since each draw command is only executed in a single GPU, CHOPIN is free of the redundant primitive projection computations that arise in traditional SFR implementations.

At group boundaries, sub-images generated by all GPUs are composed in parallel. For groups with opaque objects, sub-images can be composed out-of-order, because the pixels closer to the camera will always win. For the group of transparent objects, we take advantage of the associativity of image composition described in Section II-D: adjacent sub-images are composed asynchronously as soon as they are available.

However, naïve distribution of draw commands, such as using round-robin, can result in severe load imbalance among

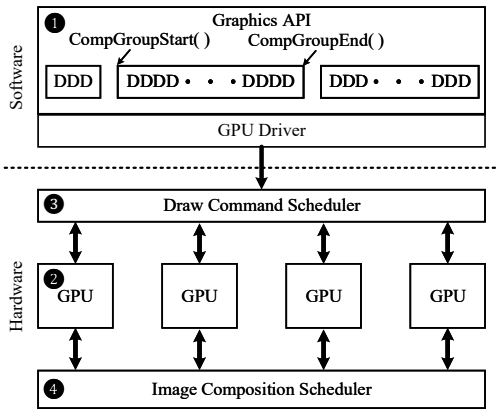


Fig. 6. High-level system overview of CHOPIN (each "D" stands for a separate draw command).

the GPUs. CHOPIN therefore includes a novel draw command scheduler (Section IV-D), which can load-balance draw commands among the GPUs based on the dynamic execution state. To mitigate network congestion and avoid unnecessary stalls, we also propose a scheduler for sub-image composition (Section IV-E), which ensures that any two GPUs can start composition only when their sub-images are ready and neither of them is composing with other GPUs.

Fig. 5 illustrates the potential of CHOPIN in an ideal system where all intermediate results are buffered on-chip and the inter-GPU links have zero latency and unlimited bandwidth: parallel image composition offers up to 1.68× speedups (1.31× gmean) over the best prior SFR solution (see Section V for evaluation methodology).

IV. The CHOPIN Architecture

The high-level system architecture of CHOPIN is shown in Fig. 6, and consists of extensions in both the software and hardware layers.

In the software layer ❶, we divide draw commands into multiple groups. At the beginning and the end of each group, we insert two new API functions *CompGroupStart()* and *CompGroupEnd()* to start and finish the image composition. We also extend the driver by implementing a separate command list for each GPU.

In the hardware layer, we connect multiple GPUs with high-speed inter-GPU links ❷, similar to NVIDIA DGX [47, 49], and present them to the OS as a single larger GPU. Draw commands issued by the driver are distributed among the different GPUs by a hardware scheduler ❸. After all draw commands of a single composition group have finished, *CompGroupEnd()* is called to compose the resulting sub-images. An image composition scheduler ❹ orchestrates which pairs of GPUs can communicate with one another at any given time.

A. Software Extensions

We first explain the semantics of extended graphics API functions. *CompGroupStart()* is called before each composition group starts: it passes the number of primitives and the transparency information to the GPU driver, which will then send these data to the GPU hardware. If there are transparent objects in composition group, the GPU driver allocates extra memory for sub-images in all GPUs, because transparent sub-images cannot be composed with the background independently. When function *CompGroupEnd()* is called, the GPU driver sends a COMPOSE command to each GPU to start the composition workflow, described in detail in Section IV-C.

The necessity of grouping draw commands is derived from the various properties of each draw command. CHOPIN assumes Immediate Mode Rendering (IMR), so we only group consecutive draw commands in a greedy fashion; however, more sophisticated mechanisms could potentially reorder draw commands to create larger composition group at the cost of additional complexity. When processing a sequence of draw commands, a group boundary is inserted between two adjacent draw commands on any of the following events:

1) swapping to the next frame,
2) switching to a new render target or depth buffer,
3) enabling or disabling updates to the depth buffer,
4) changing the fragment occlusion test function, or
5) changing the pixel composition operator.

Event 1 is straightforward, as we have to finish the current frame before moving to the next one. Render targets (RTs) are a feature that allow 3D objects to be rendered in an intermediate memory buffer, instead of the framebuffer (FB); they can be manipulated by pixel shaders in order to apply additional effects to the final image, such as light bloom, motion blur, etc. A depth buffer (or Z Buffer) is a memory structure that records the depth value of screen pixels, and is used to compute the occlusion status of newly incoming fragments. For both, Event 2 is necessary to maintain inter-RT and inter-depth-buffer dependencies, where the computing of future RTs and depth buffers depends on the content recorded in the current one.

In graphics applications, some draw commands check the depth buffer for occlusion verification without updating it. Not inserting a boundary here could allow some fragments to pass the depth test and update the frame buffer by mistake, leading to an incorrect final image. We use Event 3 to create a clean depth buffer before these draw commands begin.

Boundaries at Event 4 are needed because draw commands use depth comparison operators to retain or discard incoming fragments. Since CHOPIN distributes draw commands among multiple GPUs, having multiple comparison functions (e.g., *less-than* and *greater-than*) in a single group can scramble the depth comparison order and lead to incorrect depth verification. A group boundary at Event 4 will guarantee that every time a new comparison function is applied, the depth test will start from a correct value.

As described in Section II-D, pixel blending of consecutive draw commands is associative as long as a single blending operator (e.g., *over*) is used. However, the associativity is not transitive across different operators (e.g., mixed *over* and *additive* operators are not associative), and the composition of opaque and transparent objects also cannot be interleaved.

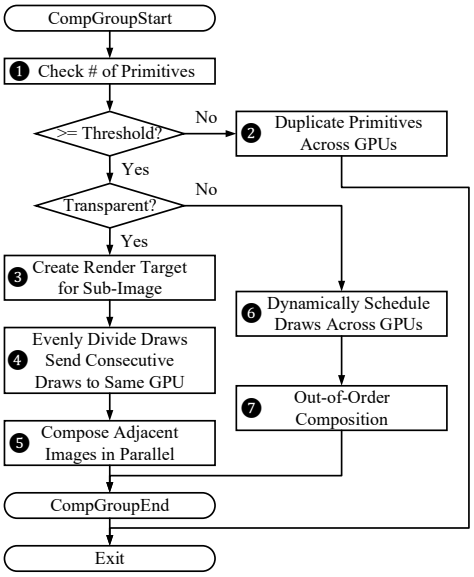


Fig. 7. The workflow of each composition group.

Hence, whenever any draw command changes to a new operator (i.e., Event 5), we create a group boundary.

B. Hardware Extensions

Besides inter-GPU communications, the main operations of image composition are (a) reading local sub-image before sending it out, and (b) composing pixels in destination GPUs. As both of these functions are carried out by the ROP, they do not require new functional components in CHOPIN.

However, since SFR (Split Frame Rendering) splits 2D screen space into multiple regions and assigns each region to a specific GPU, pixels must eventually be exchanged among GPUs after sub-images are generated, we need a hardware component that computes destination GPUs of individual pixels. We therefore slightly extend the ROPs with a simple structure that distributes pixels to different GPUs according to their screen positions.

We also require a draw command scheduler and an image composition scheduler to address the problems of load imbalance and network congestion, which are two main performance bottlenecks of a naïve implementation of CHOPIN. We describe them in Section IV-D and IV-E, respectively.

C. Composition Workflow

Fig. 7 shows the workflow of each composition group. When a composition group begins, we first check how many primitives (e.g., triangles) are included in this group ❶. If the number of primitives is smaller than a certain threshold, we revert to traditional SFR and duplicate all primitives in each GPU ❷. This is a tradeoff between redundant geometry processing and image composition overhead. For example, some draw commands are executed to set up the background before real objects are rendered; because these draw commands simply cut a rectangle screen into two triangles, the geometry processing

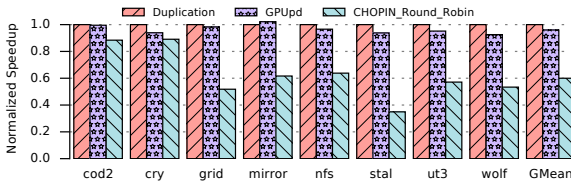


Fig. 8. Performance overhead of round-robin draw command scheduling (normalized to the system which duplicates all primitives across GPUs).

overhead is much smaller than other graphics pipeline stages, and the overhead of redundant geometry processing is also much smaller than the cost of image composition. Although this threshold is an additional parameter that must be set, our sensitivity analysis results (see Fig. 22) show that the threshold value does not substantially impact the performance, so this is not a significant concern.

For each composition group that warrants parallel image composition, we first check whether the group contains transparent objects. If so, the GPU driver needs to create extra render targets for sub-images in each GPU ❸. This is necessary because transparent objects cannot be merged with the background until all sub-images have been composed — otherwise, the background pixels will be composed multiple times, creating an incorrect result. To protect the input order of transparent primitives and achieve reasonable load balance at the same time, we evenly divide draw commands and simply distribute the same amount of continuous primitives across GPUs ❹. While more complex solutions exist, this simple workload distribution is acceptable because, in current applications, only a small fraction of draw commands are transparent. After a GPU has finished its workload, we can begin to compose adjacent sub-images asynchronously by leveraging associativity ❺.

If the group has no transparent objects, CHOPIN dynamically distributes draw commands with our proposed scheduler ❻; in this case, it is not necessary to create extra render targets because generated sub-images will overwrite the background anyway. Finally, opaque sub-images are composed out-of-order ❼ by simply comparing their distances to the camera (depth value); sub-image pixels which are closer to the camera will be retained for final image composition.

D. The Draw Command Scheduler

Although the parallel image composition technique in CHOPIN can avoid sequential inter-GPU synchronizations, the correct final image can only be generated after all sub-images have been composed; therefore, the slowest GPU will determine the overall system performance. As Fig. 8 shows, simple draw command scheduling, such as round-robin, can lead to severe load imbalance and substantially impact performance.

To achieve optimal load balance, we would ideally like to know the exact execution time of each draw command; however, this is unrealistic before the draw command is actually executed. Therefore, we need to approximately predict the draw command running time. A complete heuristic for rendering time estimation has been proposed in [62]: $t = c_1 \times \#tv + c_2 * \#pix$,

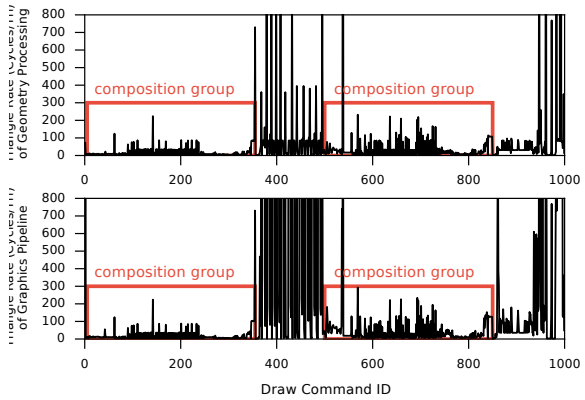


Fig. 9. Triangle rate of geometry processing stage (top) and whole graphics pipeline (bottom). Data is from cod2, other applications have the same trend.

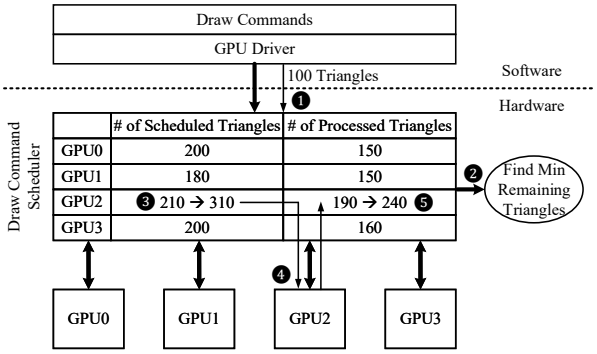
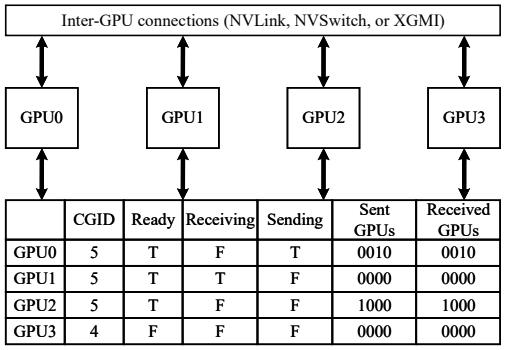


Fig. 10. Draw command scheduler microarchitecture.

where t is the estimated rendering time, *#tv* is the number of transformed vertices, *#pix* is the number of rendered pixels, and c_1 and c_2 are the vertex rate and pixel rate. Although this heuristic considers both geometry and fragment processing stages, the value of c_1 and c_2 can change dramatically across draw commands, and we cannot use this approach directly. OO-VR [65] samples these parameters on the first several draw commands and uses them for the remainder of the rendering computation; however, we have found that these parameters vary substantially, and such samples form a poor estimate for the dynamic execution state of the whole system. Other prior work [10] instead uses the triangle count of each draw command (which can be acquired from applications) as a heuristic to estimate rendering time. However, dynamically keeping tracking of all triangles throughout the graphics pipeline is complicated, especially after a triangle is rasterized into multiple fragments.

Fortunately, as Fig. 9 shows, the *triangle rate* (i.e., cycles/triangle) of the geometry processing stage is similar to that of the whole graphics pipeline — this is similar to how the instruction processing rate in a CPU frontend limits the performance of the CPU backend. We therefore propose to use the number of remaining triangles in the geometry processing stage as an estimate of each GPU's remaining workload. Every time a draw command is issued by the GPU driver, we simply



	CGID	Ready	Receiving	Sending	Sent GPUs	Received GPUs
GPU0	5	T	F	T	0010	0010
GPU1	5	T	T	F	0000	0000
GPU2	5	T	F	F	1000	1000
GPU3	4	F	F	F	0000	0000

Fig. 11. Image composition scheduler microarchitecture.

Field	Meaning
CGID	Composition Group ID
Ready	Ready to compose with others?
Receiving	Receiving pixels from another GPU?
Sending	Sending pixels to another GPU?
SentGPUs	GPUs the sub-image has been sent to
ReceivedGPUs	GPUs we have composed with

TABLE I
FIELDS IN THE IMAGE COMPOSITION SCHEDULER.

distribute it to the GPU which has the fewest remaining triangles in geometry processing stage.

The microarchitecture of our draw command scheduler is shown in Fig. 10. The main structure is a table, in which each GPU has an entry to record the number of scheduled and processed triangles in that GPU; the remaining triangle count is the difference. The scheduled triangle count increments when a draw command is scheduled to a GPU, while the processed count increments as triangles finish geometry processing.

Fig. 10 also shows a running example of how the scheduler operates. First, the GPU driver issues a draw command with 100 triangles ❶. Next, the draw command scheduler finds that GPU2 currently has the fewest remaining triangles ❷. The triangle count of this draw command is therefore added to the number of triangles scheduled to GPU2 ❸, and the scheduler distributes this draw command to GPU2 ❹. Once the triangles pass through the geometry processing stage in graphics pipeline, the number of processed triangles for GPU2 is increased accordingly ❺.

E. Image Composition Scheduler

Once each GPU has finished its workload, it can begin to communicate with other GPUs for sub-image composition. However, blind inter-GPU communication can result in the congestion and under-utilization of interconnect resources (see Section II-D). The most straightforward scheme, direct-send, sends the screen regions to any other GPUs without knowing if the destination GPU can accept it; if the target GPU is still computing, the waiting inter-GPU messages will block the interconnect. For example, assuming a situation where all GPUs except GPU0 have finished their draw commands, so

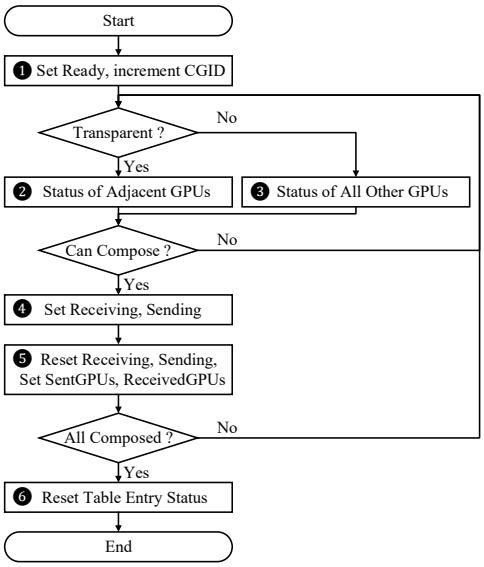


Fig. 12. Image composition scheduler workflow.

GPUs begin to send their sub-images to GPU0. Because GPU0 is still running, inter-GPU messages will be blocked in the network. Even though GPUs could have communicated with another GPU rather than GPU0, now they have to wait until GPU0 is able to drain the network. Therefore, an intelligent scheduling mechanism for image composition is necessary.

Our proposed composition scheduler, shown in Fig. 11, aims to avoid stalls due to the GPUs that are still running their scheduled draw commands or busy composing with other GPUs. It records the composition status (Table I) of each GPU in a table: the composition group ID (CGID) is used to distinguish different groups, the Ready flag is set while a GPU generated its sub-image and became ready to compose with others, and the Receiving and Sending flags are used to indicate that a GPU is busy exchanging pixels with another GPU. Finally, SentGPUs and ReceivedGPUs record the GPUs with which a GPU has already communicated in a bit vector, vector size is same as the number of GPUs in the system.

Fig. 12 shows the image composition scheduler workflow. Once a GPU has finished all draw commands and generated a sub-image, we set its Ready flag and increment the CGID by one to start a new composition phase ❶. We then check the status of other GPUs to see if any available GPUs can compose with each other. For groups of transparent objects ❷, only adjacent GPUs are checked because transparent sub-images cannot be composed entirely out-of-order (Section II-D); for opaque groups, all GPUs are checked❸. Composition starts only if the remote GPU (1) is ready to compose and running in the same composition group (i.e., CGIDs are same), (2) has not yet been composed with (i.e., not set in ReceivedGPUs), and (3) is not sending pixels to another GPU.

As an example, consider the status in Fig. 11. We can see that GPU0 and GPU2 have composed with each other, GPU3 is still running, and GPU1 just finished its workload and set its Ready flag. At this moment, GPU1 can compose with GPU0, so we set the Receiving flag of GPU1 and the Sending flag of GPU0 to indicate that these two GPUs are busy ❹. When image composition starts, GPU0 will read its sub-image and send out the region corresponding to GPU1. After these two GPUs have finished composition, we will reset the Receiving flag of GPU1 and the Sending flag of GPU0. Concurrently, we will also add GPU0 into the ReceivedGPUs field of GPU1 and add GPU1 into the SentGPUs field of GPU0 ❺. This procedure is repeated until all sub-images are composed. Finally, we reset the table entry after a GPU has sent its sub-image to all other GPUs and the sub-images of all other GPUs has also been received ❻. The composition is finished once each GPU has composed with all other GPUs and, for transparent sub-images, the background.

F. Scheduler Implementation

In CHOPIN, both the draw command scheduler and image composition scheduler are centralized, and can be easily implemented on interconnects like NVSwitch [43], already widely used in modern multi-GPU platforms [47, 49, 50]. Because the hardware status and scheduling information need to be exchanged only infrequently and require little bandwidth, scalability is not an issue (see Section VI-D).

In this paper, we opt to implement both schedulers in hardware. This is because they rely on dynamic execution state information — such as the current remaining workload and the current busy/working status for each GPU — that is not available in host-side software. Nevertheless, it also would be possible to expose this hardware state information to software and manage scheduling in software (e.g., in the GPU's command processor). Indeed, our implementation serves as a proof-of-concept for such schedulers. We leave the exploration and analysis of the full scheduler design space to future work.

V. Methodology

We evaluate CHOPIN by extending ATTILA [19, 39], a cycle-level GPU simulator which implements a wide spectrum of graphics features present in modern GPUs. Unfortunately, the latest ATTILA is designed to model an AMD TeraScale2 architecture [33], and cannot be configured to reflect the latest NVIDIA Volta [48] or Turing [44] systems; therefore, to fairly simulate the performance of different SFR implementations, we scale down system parameters, such as the number of SMs and ROPs, accordingly (Table II). Similar simulation strategies have been widely used in related prior work [63, 64, 65, 66]. We extend the GPU driver for issuing draw commands and hardware register values to different GPUs. Similar to the existing NVIDIA DGX system [47, 49], we model the inter-GPU links with point-to-point connections between GPU pairs, with a default bandwidth and latency of 64GB/s and 200 cycles.

As benchmarks, we use eight single-frame traces as shown in Table III, which we manually annotate to insert the new API functions *CompGroupStart()* and *CompGroupEnd()* at

Structure	Configuration
GPU frequency	1GHz
Number of GPUs	8
Number of SMs	64 (8 per GPU)
Number of ROPs	64 (8 per GPU)
SM configurations	32 shader cores per SM 4 texture units
L2 Cache	6MB in total
DRAM	2TB/s, 8 channels 8 banks per channel
Composition group # primitives threshold	4096
Inter-GPU bandwidth	64GB/s (uni-directional)
Inter-GPU latency	200 cycles

TABLE II
SIMULATED ARCHITECTURE CONFIGURATIONS.

Benchmark	Abbr.	Resolution	# Draws	# Triangles
Call of Duty 2	cod2	640×480	1005	219,950
Crysis	cry	800×600	1427	800,948
GRID	grid	1280×1024	2623	466,806
Mirror's Edge	mirror	1280×1024	1257	381,422
Need for Speed: Undercover	nfs	1280×1024	1858	534,121
S.T.A.L.K.E.R.: Call of Pripyat	stal	1280×1024	1086	546,733
Unreal Tournament 3	ut3	1280×1024	1944	630,302
Wolfenstein	wolf	640×480	1697	243,052

TABLE III
BENCHMARKS USED FOR EVALUATION.

composition group boundaries. All of the benchmarks come from the real-world games; the number of draw commands and the number of primitives (triangles) and the target resolutions vary across the set, and are shown in Table III.

Our SFR implementation splits each frame by interleaving 64×64 pixel tiles to different GPUs. Unlike AFR, SFR needs to handle the read-after-write dependencies on render targets and depth buffers. To ensure memory consistency, every time the application switches to a new render target or depth buffer, our simulation invokes an inter-GPU synchronization which requires each GPU to broadcast the latest content of its current render targets and depth buffers to other GPUs.

Apart from our CHOPIN system, we implement primitive duplication, which we use as the baseline. We also implement the best prior work GPUpd [28], modelling both optimizations: batching and runahead execution[1]. To explore the upper bound on the performance of each technique, we also idealize GPUpd and CHOPIN in the same way: unlimited on-chip memory for buffering intermediate results, zero inter-GPU latency, and infinite inter-GPU bandwidth.

VI. EVALUATION RESULTS

In this section, we first compare the performance of CHOPIN, primitive duplication, and GPUpd; we then explore the design space via sensitivity analysis, and evaluate the hardware costs.

A. Performance Analysis

The overall performance of multiple SFR implementations is shown in Fig. 13. The performance of GPUpd is comparable

[1]We created a best-effort realistic implementation of GPUpd as well as an idealized variant.

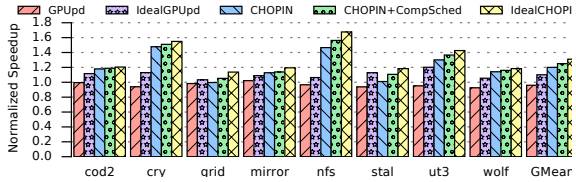


Fig. 13. Performance of an 8-GPU system, baseline is primitive duplication with configurations of Table II. (CompSched: composition scheduler)

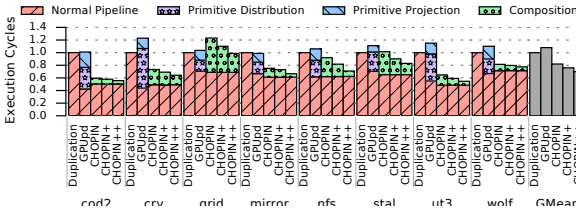


Fig. 14. Execution cycle breakdown of graphics pipeline stages, normalize all results to the cycles of primitive duplication. (CHOPIN+: CHOPIN + composition scheduler, CHOPIN++: IdealCHOPIN)

to conventional primitive duplication. Idealizing GPUpd (i.e., our best implementation of GPUpd) can slightly improve the performance, but it remains substantially worse than CHOPIN. With the image composition scheduler enabled, CHOPIN works 25% (up to 56%) better than primitive duplication, and only 4.8% slower than IdealCHOPIN.

Fig. 14 shows that the performance improvement of CHOPIN comes mainly from the reduced synchronization overheads: for GPUpd, this is the extra primitive projection and distribution stages, while for CHOPIN this is the image composition stage (e.g., the composition overhead of `grid` is large because it has much bigger inter-GPU traffic load, see Fig. 17). Conventional primitive duplication suffers because of redundant geometry processing, which CHOPIN entirely avoids. Even though GPUpd still performs some redundant computation in the primitive projection stage, sequential inter-GPU primitive exchange is its critical performance bottleneck.

CHOPIN avoids redundant geometry processing by distributing each draw command to a specific GPU, and substantially reduces the overhead of inter-GPU synchronization through parallel composition. With the image composition scheduler, the composition cost is reduced even more by avoiding unnecessary network congestion.

B. Impact on the Depth/Stencil Test

The performance of CHOPIN comes at the cost of slightly increasing fragment processing overheads: because sub-images are composed in parallel, some fragments that would be depth-culled in a single GPU may still be processed because a GPU in CHOPIN may not have the relevant occluding fragment. However, we assume immediate-mode rendering (IMR), where draw commands cannot be reordered. Programmers generally arrange draw commands front-to-back to facilitate depth culling, and this order is retained by CHOPIN/IMR. The performance

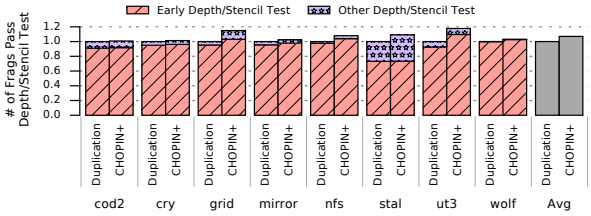


Fig. 15. Number of fragments that pass depth/stencil tests in an 8-GPU system, normalized to primitive duplication. (GPUpd is a sort-first mechanism, so its depth/stencil test results are same as primitive duplication. CHOPIN+: CHOPIN + composition scheduler)

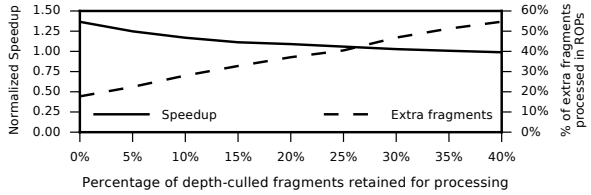


Fig. 16. Performance sensitivity (solid line) to the additional fragment workload (dashed line) caused by artificially reducing depth-culling rates (`ut3`, 8 GPUs). Both normalized to the frame duplication baseline.

impact of the additional fragment processing overhead is therefore small (Fig. 15).

The early depth/stencil test is an important optimization supported by most modern GPUs: fragments that do not pass this test are culled and never passed to the shader. Fig. 15 shows that most fragments in our benchmarks that survive to the shading step were subjected to (and passed) the early depth/stencil test[2]. As Fig. 15 shows, CHOPIN increases the number of fragments that pass (i.e., the additional work) only slightly: the total number of fragments processed in ROPs increases by only 3%, 5.4%, and 7.1% on average in systems of 2, 4, and 8 GPUs (with 18% in the worst case for `ut3` on 8 GPUs), which still permits overall CHOPIN speedups of up to 1.56×. Because triangle counts grow much faster than resolutions, the overall performance impact of depth/stencil test will be even more limited in new workloads, which have millions to billions of tiny triangles with sizes often smaller than pixels [8].

To model hypothetical workloads with even more reduced culling effectiveness, we also ran CHOPIN on `ut3` while artificially inflating this fragment overhead. To do this, we randomly retained a fixed percentage of depth-culled fragments, and processed them in the rest of the fragment pipeline as if they had not been culled. Fig. 16 shows that this increases the number of processed fragments, and slightly impacts performance; however, we needed to retain nearly half of all culled fragments to negate the performance benefits of CHOPIN. This is much higher than what occurs in real workloads, where most fragments are still depth-culled (because CHOPIN retains the front-to-back order within each GPU), and primitive counts grow faster than resolutions.

[2]In general, the early depth/stencil test is sometimes disabled so that the fragment shader can discard fragments or overwrite their depth values.

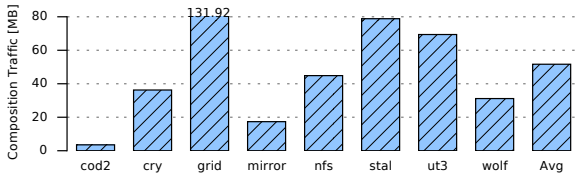


Fig. 17. Traffic load of parallel image composition.

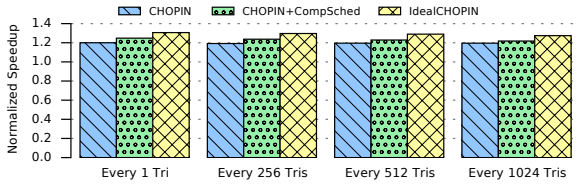


Fig. 18. Performance sensitivity to the update frequency of draw command scheduler (baseline is primitive duplication with configurations from Table II).

C. Composition Traffic Load

To reduce network traffic, CHOPIN only exchanges screen regions assigned to the GPUs that are communicating at any given moment. We also filter out the screen tiles that are not rendered by any draw commands, as they do not need to be composed. As Fig. 17 shows, the average traffic load of image composition is only 51.66MB. Fig. 14 shows that this does not create a substantial execution overhead, especially with the image composition scheduler enabled. The large traffic load in `grid` is due to many large triangles that cover big screen regions; we leave optimizing this to future work.

D. Scalability of Schedulers

Draw command scheduler. In most of our experiments, we allowed the GPUs to update the draw command scheduler statistics for every triangle processed, creating an average of 1.7MB traffic with 4B message size. To account for scaling to much larger systems and much larger triangle counts, however, we also investigated how a larger update interval would affect the performance of CHOPIN. Fig. 18 sweeps this update frequency from every triangle to every 1024 triangles on an otherwise identical system; the average performance improvement of CHOPIN drops very slightly from 1.25× to 1.22×. With updates every 1024 triangles and 4B messages, the total update traffic load would be 4KB for 1 million triangles and 4MB for 1 billion triangles.

Image composition scheduler. The image composition scheduler receives notifications from GPUs at composition boundaries that they are ready to accept work, and sends notifications back to GPUs — 7 requests and 7 responses for each GPU in an 8-GPU system, plus an 8th pair to compose with the background — which results in $(8+8)\times 8\times 4 = 512$B with 4B messages. The bandwidth cost of both schedulers is negligible compared to sub-image frame content.

E. Sensitivity Analysis

To understand the relationship between our architectural parameters and the performance of CHOPIN, we performed sensitivity

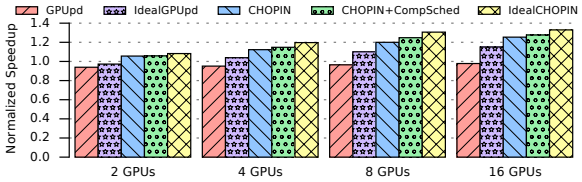


Fig. 19. Performance sensitivity to number of GPUs (for each GPU count configuration, baseline is primitive duplication with the same GPU count and other settings as in Table II).

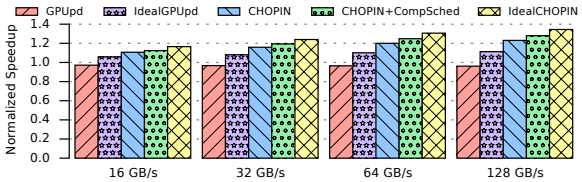


Fig. 20. Performance sensitivity to inter-GPU link bandwidth (baseline is primitive duplication with configurations of Table II).

studies across a range of design space parameters.

GPU count. Even though integrating more GPUs in a system can provide abundant resources to meet the constantly growing computing requirements, it also can impose bigger challenge on inter-GPU synchronizations. As Fig. 19 shows, GPUpd is constrained by the sequential primitive distribution, and performance does not scale with GPU count. In contrast, because CHOPIN parallelizes image composition, the inter-GPU communication is also accelerated with more GPUs. Therefore, the performance of CHOPIN is scalable and the improvement versus prior SFR solutions grows as the number of GPUs increases. Meanwhile, the image composition scheduler becomes more effective when the GPU count is higher: this is because naïve inter-GPU communication for image composition can congest the network more frequently with more GPUs, which is a bigger bottleneck for a larger system.

Inter-GPU link bandwidth and latency. Since inter-GPU synchronization relies on the inter-GPU interconnect, we investigated sensitivity to link bandwidth and latency. CHOPIN performance scales with bandwidth (Fig. 20), unlike GPUpd. Similarly, CHOPIN is not significantly affected by link latency (Fig. 21), unlike GPUpd where latency quickly bottlenecks sequential primitive exchange.

Composition group size threshold. This parameter makes a tradeoff between the redundant geometry processing and the image composition overhead: if the number of primitives inside a composition group is smaller than a specific threshold, CHOPIN reverts to primitive duplication (see Fig. 7). In theory, this threshold could be important: if set too low, it might not filter out most composition groups with few primitives, and if set too high, we could lose the potential performance improvement of parallel image composition. However, as Fig. 22 shows, it turns out that the performance of CHOPIN is not very sensitive to the configuration of this threshold value, and this setting should be of little concern to programmers.

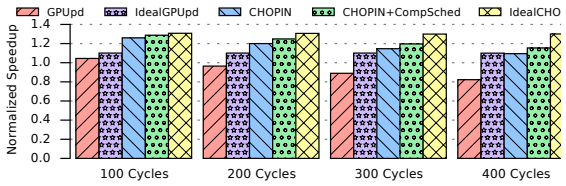


Fig. 21. Performance sensitivity to inter-GPU link latency (baseline is primitive duplication with configurations of Table II).

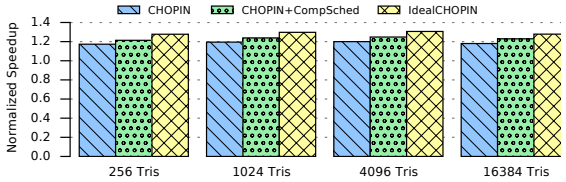


Fig. 22. Performance sensitivity to the threshold of composition group size (baseline is primitive duplication with configurations of Table II).

The main reason for the lack of sensitivity is that the statistic distribution of composition group sizes is bimodal: most composition groups have either a large number of triangles (e.g., consecutive draw commands for objects rendering) or very few triangles (e.g., background draw commands), and many threshold settings will separate them. For example, if the threshold is set as 4,096, CHOPIN will accelerate 6.5 composition groups on average, covering 92.44% of the triangles in the entire application; increasing the threshold to 16,384 will accelerate 5.25 composition groups, covering 89.83% triangles on average.

F. Hardware Costs

The draw command scheduler and image composition scheduler constitute the main hardware cost of the CHOPIN system. In an 8-GPU system, both schedulers have 8 entries. Each entry in the draw command scheduler has two fields: the number of scheduled triangles and the number of processed triangles. To cover the requirements of most existing and future applications, we conservatively allocate 64 bits for each field. Therefore, the total size of draw command scheduler is 128 bytes.

As discussed in Section VI-E, with a group size threshold of 4,096, up to 13 (6.5 on average) draw command groups will trigger image composition, so we assume one byte is enough to represent a CGID for the image composition scheduler. The Ready, Receiving and Sending flags are all single bits. SentGPUs and ReceivedGPUs are bit vectors with as many bits as the number of GPUs in the system (for us, one byte). Therefore, the total size of the image composition scheduler in our implementation is 27 bytes.

G. Scaling to Modern and Future Games

Since the ATTILA simulation framework only supports an older API (DirectX 9), we cannot directly evaluate very recent workloads. We therefore used profiling in a commercial GPU to determine how well sort-last rendering approaches like CHOPIN might scale to future games.

The key question is which part of the rendering pipeline grows faster: fragment processing (because resolution increases) or primitive processing (because the level of detail increases). If fragment counts dominate growth, sort-first approaches like GPUpd are better, since they do extra work in the primitive processing phase but no extra work in the fragment processing phase. If, on the other hand, primitive processing grows faster, sort-last approaches like CHOPIN are better, since they do not add any work to the primitive processing step.

We profiled two workloads from 2017 and 2020 using NVIDIA Nsight [52] on an NVIDIA GTX 1060 GPU: the Unigine Superposition benchmark from 2017 [61] and Crysis Remastered [22], a game released in September 2020. Crysis Remastered has an average of 12 million triangles per frame, with an average primitive processing time of 11.57ms and an average fragment processing time of 11.11ms, even in 1080p — this is in contrast to 4 million triangles for Superposition, and fewer than 1 million triangles for the older benchmarks we used in this paper.

This shows that primitive processing is already outstripping fragment processing. Indeed, primitive counts look likely to scale dramatically: the Unreal Engine 5 announcement envisions a billion triangles per frame in the near future [8]. These trends favour sort-last schemes like CHOPIN.

H. Massive Multi-GPU Systems

As is, CHOPIN is applicable to NVIDIA DGX-scale systems [47, 49]. Systems which are significantly larger (e.g., 1024 GPUs) can potentially reduce the benefit of CHOPIN, because each GPU will only get very few draw commands and will process more unnecessary fragments. However, it's not quite realistic to render single frames with 1024 GPUs, as it can result in severe hardware underutilization. We believe large-scale systems may need more complicated rendering mechanisms, such as the combination of AFR and SFR. We leave this to future work.

VII. Related Work

Multi-GPU Systems. The latest work in multi-GPU systems is discussed in Section II-C. Most focuses on general-purpose applications, rather than graphics rendering like CHOPIN. GPUpd [28] and OO-VR [65] are two multi-GPU proposals that attempt to leverage modern, high-speed inter-GPU connections. However, as discussed in Section III-A, GPUpd is bottlenecked by a sequential inter-GPU primitive exchange step, while CHOPIN composes sub-images in parallel. OO-VR is a rendering framework to improve data locality in VR applications, orthogonal to the problem of efficient image composition for SFR that we address in this paper. Unlike OO-VR, the draw command distribution in CHOPIN does not rely on statically computed parameters; CHOPIN also includes an image composition scheduler to make full use of the available network resources.

NVIDIA's SLI [45] attempts to balance the workload by dynamically adjusting how the screen is divided among GPUs. However, it still duplicates all primitives in every GPU, and incurs the attendant overheads. Both DirectX 12 [6] and Vulkan [4] expose multi-GPU hardware to programmers via API, but relying only on this would require programmers to have exact static knowledge of the workload (e.g., workload distribution). CHOPIN can simplify programming and deliver reliable performance through dynamic scheduling in hardware.

Parallel Rendering. Most SFR mechanisms were originally targeted at PC clusters. Among these, WireGL [26], Chromium [27], and Equalizer [20] are high-level APIs which can allocate workload among machines based on different configurations. However, when the system is configured as sort-first, they use CPUs to compute the destinations of each primitive, and performance is limited by the poor computation throughput of CPUs. When the system is configured as sort-last, they assign one specific machine to collect all sub-images from others for composition, which again creates a bottleneck. In contrast, CHOPIN distributes draw commands to different GPUs based on dynamic execution state, and all GPUs in the system contribute to image composition in parallel.

To accelerate image composition, PixelFlow [37] and Pixel-Planes 5 [23] implemented application-specific hardware, with significant overheads. In contrast, CHOPIN takes advantage of existing multi-GPU systems and high-performance inter-GPU links, and incurs very small hardware costs. RealityEngine [9] and Pomegranate [21] aim to improve load balancing by frequently exchanging data before geometry processing, before rasterization, and after fragment processing; however, these complicated synchronization patterns are hard to coordinate, and huge traffic loads can be imposed on the inter-GPU links.

Single-GPU Systems. Besides parallel rendering, significant work has also been done on graphics processing in single GPU or mobile GPU systems. By leveraging the similarity between consecutive frames, Arnau et al. proposed fragment memorization to filter out redundant fragment computation [14]. Rendering Elimination [13] shares the same observation of similarity, but excludes redundant computation at the coarser granularity of screen tiles. To verify fragment occlusion as early as possible, Anglada et al. proposed early visibility resolution, a mechanism that leverages the visibility information obtained from a frame to predict the visibility of the next frame [12]. Texture data consumes significant off-chip memory bandwidth, so Xie et al. explored the use of processing-in-memory to reduce texture memory traffic [64]. In contrast to all these efforts, CHOPIN focuses on the efficient inter-GPU synchronization of parallel rendering in multi-GPU systems.

VIII. Conclusion

In this paper, we introduce CHOPIN, a novel architecture for split frame rendering in multi-GPU systems. CHOPIN is a sort-last rendering scheme which distributes each draw command to a unique GPU and avoids redundant geometry processing. By leveraging the mathematical properties of image composition and modern high-speed inter-GPU links, CHOPIN can perform composition in parallel without requiring sequential inter-GPU communication.

CHOPIN includes a draw command scheduler to address load imbalance, and an image composition scheduler to reduce network congestion. Overall, CHOPIN outperforms the best prior work by up to 1.56× (1.25× gmean), and — in contrast to existing solutions — scales as the number of GPUs grows.

IX. Acknowledgments

The authors are grateful to the anonymous reviewers for their insightful feedback. This research was supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] "AMD's answer to Nvidia's NVLink is xGMI, and it's coming to the new 7nm Vega GPU," https://www.pcgamesn.com/amd-xgmi-vega-20-gpu-nvidia-nvlink, accessed on 2020-07-31.

[2] "How to fix micro stutter in videos+games?" https://www.reddit.com/r/nvidia/comments/3su8qq/how_to_fix_micro_stutter_in_videosgames/, 2016, accessed on 2020-07-31.

[3] "Nvidia GeForce GTX 1080i review: The best 4K graphics card right now," https://www.rockpapershotgun.com/2018/01/30/nvidia-geforce-gtx-1080-review-best-4k-graphics-card/, 2018, accessed on 2020-07-31.

[4] "Vulkan 1.1 out today with multi-GPU support, better DirectX compatibility," https://arstechnica.com/gadgets/2018/03/vulkan-1-1-adds-multi-gpu-directx-compatibility-as-khronos-looks-to-the-future/, 2018, accessed on 2020-07-31.

[5] "What is microstutter and how do I fix it?" https://www.pcgamer.com/what-is-microstutter-and-how-do-i-fix-it/, 2018, accessed on 2020-07-31.

[6] "Direct3D 12 Programming Guide: Multi-adapter systems," https://docs.microsoft.com/en-us/windows/win32/direct3d12/multi-engine, 2019, accessed on 2020-07-31.

[7] "How to fix stuttering of CrossFire?" https://www.reddit.com/r/crossfire/comments/ddlcl9/how_to_fix_stuttering_of_crossfire/, 2019, accessed on 2020-07-31.

[8] "A first look at Unreal Engine 5," https://www.unrealengine.com/en-US/blog/a-first-look-at-unreal-engine-5, 2020, accessed on 2020-07-31.

[9] K. Akeley, "Reality Engine Graphics," in *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*. ACM, 1993, pp. 109–116.

[10] D. G. Aliaga and A. Lastra, "Automatic Image Placement to Provide A Guaranteed Frame Rate," in *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, 1999, pp. 307–316.

[11] AMD, "AMD CrossFire guide for Direct3D® 11 applications," https://gpuopen-librariesandsdks.github.io/doc/AMD-CrossFire-guide-for-Direct3D11-applications.pdf, accessed on 2020-07-31.

[12] M. Anglada, E. de Lucas, J.-M. Parcerisa, J. L. Aragón, and A. González, "Early Visibility Resolution for Removing Ineffectual Computations in the Graphics Pipeline," in *Proceedings of the 25th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 635–646.

[13] M. Anglada, E. de Lucas, J.-M. Parcerisa, J. L. Aragón, P. Marcuello, and A. González, "Rendering Elimination: Early Discard of Redundant Tiles in the Graphics Pipeline," in *Proceedings of the 25th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 623–634.

[14] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Eliminating Redundant Fragment Shader Executions on a Mobile GPU via Hardware Memoization," in *Proceeedings of the 41th International Symposium on Computer Architecture (ISCA)*. ACM, 2014, pp. 529–540.

[15] A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, C.-J. Wu, and D. Nellans, "MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability," in *Proceeedings of the 44th International Symposium on Computer Architecture (ISCA)*. ACM, 2017, pp. 320–332.

[16] T. Baruah, Y. Sun, A. Dinçer, M. S. A. Mojumder, J. L. Abellán, Y. Ukidave, A. Joshi, N. Rubin, J. Kim, and D. Kaeli, "Griffin: Hardware-Software Support for Efficient Page Migration in Multi-GPU Systems," in *Proceedings of the 26th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 596–609.

[17] E. W. Bethel, H. Childs, and C. Hansen, *High Performance Visualization: Enabling Extreme-Scale Scientific Insight (Chapter 5)*. CRC Press, 2012.

[18] C. Boyd, "DirectX 11 Compute Shader," https://docplayer.net/36909319-Directx-11-compute-shader-chas-boyd-architect-windows-desktop-and-graphics-technology-microsoft.html, 2008, accessed on 2020-07-31.

[19] V. M. Del Barrio, C. González, J. Roca, A. Fernández, and E. Espasa, "ATTILA: A Cycle-Level Execution-Driven Simulator for Modern GPU Architectures," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2006, pp. 231–241.

[20] S. Eilemann, M. Makhinya, and R. Pajarola, "Equalizer: A Scalable Parallel Rendering Framework," *IEEE transactions on visualization and computer graphics (TVCG)*, vol. 15, no. 3, pp. 436–452, 2009.

[21] M. Eldridge, H. Igehy, and P. Hanrahan, "Pomegranate: A Fully Scalable Graphics Architecture," in *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRPAH)*. ACM, 2000, pp. 443–454.

[22] Epic Games, "Crysis Remastered," https://www.epicgames.com/store/en-US/product/crysis-remastered/home, 2010, accessed on 2020-11-11.

[23] H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel, "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories," *Siggraph Computer Graphics*, vol. 23, no. 3, pp. 79–88, 1989.

[24] D. R. Hower, B. A. Hechtman, B. M. Beckmann, B. R. Gaster, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous-Race-Free Memory Models," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2014, pp. 427–440.

[25] W. M. Hsu, "Segmented Ray Casting for Data Parallel Volume Rendering," in *Proceedings of the 1993 symposium on Parallel Rendering*. IEEE, 1993, pp. 7–14.

[26] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan, "WireGL: A Scalable Graphics System for Clusters," in *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*. ACM, 2001, pp. 129–140.

[27] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski, "Chromium: A Stream-Processing Framework for Interactive Rendering on Clusters," *ACM transactions on graphics (TOG)*, vol. 21, no. 3, pp. 693–702, 2002.

[28] Y. Kim, J.-E. Jo, H. Jang, M. Rhu, H. Kim, and J. Kim, "GPUpd: A Fast and Scalable Multi-GPU Architecture Using Cooperative Projection and Distribution," in *Proceedings of the 50th International Symposium on Microarchitecture (MICRO)*. ACM, 2017, pp. 574–586.

[29] S. Laine and T. Karras, "High-Performance Software Rasterization on GPUs," in *Proceedings of the SIGGRAPH Symposium on High Performance Graphics (HPG)*. ACM, 2011, pp. 79–88.

[30] D. Luebke and G. Humphreys, "How GPUs Work," *Computer*, vol. 40, no. 2, pp. 96–100, 2007.

[31] D. Lustig, S. Sahasrabuddhe, and O. Giroux, "A Formal Analysis of the NVIDIA PTX Memory Consistency Model," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2019, pp. 257–270.

[32] K.-L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh, "A Data Distributed, Parallel Algorithm for Ray-Traced Volume Rendering," in *Proceedings of Parallel Rendering Symposium*. IEEE, 1993, pp. 15–22.

[33] Mike Houston, "Anatomy of AMD's TeraScale Graphics Engine," http://attila.ac.upc.edu/wiki/images/3/34/Houston-amd-terascale.pdf, 2008, accessed on 2020-07-31.

[34] U. Milic, O. Villa, E. Bolotin, A. Arunkumar, E. Ebrahimi, A. Jaleel, A. Ramirez, and D. Nellans, "Beyond the Socket: NUMA-Aware GPUs," in *Proceedings of the 50th International Symposium on Microarchitecture (MICRO)*. ACM, 2017, pp. 123–135.

[35] L. Moll, A. Heirich, and M. Shand, "Sepia: Scalable 3D Compositing Using PCI Pamette," in *Proceedings of the 7th International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 1999, pp. 146–155.

[36] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, "A Sorting Classification of Parallel Rendering," *IEEE Computer Graphics and Applicationsi (CG&A)*, vol. 14, no. 4, pp. 23–32, 1994.

[37] S. Molnar, J. Eyles, and J. Poulton, "PixelFlow: High-Speed Rendering Using Image Composition," in *Proceedings of the 19th Annual Conference*

on Computer Graphics and Interactive Techniques (SIGGRAPH). ACM, 1992, pp. 231–240.

[38] J. R. Monfort and M. Grossman, "Scaling of 3D Game Engine Workloads on Modern Multi-GPU Systems," in *Proceedings of the Conference on High Performance Graphics (HPG)*. ACM, 2009, pp. 37–46.

[39] V. Moya, C. Gonzalez, J. Roca, A. Fernandez, and R. Espasa, "Shader Performance Analysis on A Modern GPU Architecture," in *Proceedings of the 38th International Symposium on Microarchitecture (MICRO)*. IEEE, 2005, pp. 355–364.

[40] C. Mueller, "The Sort-First Rendering Architecture for High-Performance Graphics," in *Proceedings of the 1995 symposium on Interactive 3D graphics (I3D)*, 1995, pp. 75–84.

[41] U. Neumann, "Communication Costs for Parallel Volume-Rendering Algorithms," *IEEE Computer Graphics and Applicationsi (CG&A)*, vol. 14, no. 4, pp. 49–58, 1994.

[42] NVIDIA, "NVIDIA NVLink: High Speed GPU Interconnect," https://www.nvidia.com/en-us/design-visualization/nvlink-bridges/, accessed on 2020-07-31.

[43] NVIDIA, "NVIDIA NVSwitch: The World's Highest-Bandwidth On-Node Switch," https://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf, accessed on 2020-07-31.

[44] NVIDIA, "NVIDIA Turing GPU Architecture Whitepaper," https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf, accessed on 2020-07-31.

[45] NVIDIA, "SLI Best Practices," http://developer.download.nvidia.com/whitepapers/2011/SLI_Best_Practices_2011_Feb.pdf, 2011, accessed on 2020-07-31.

[46] NVIDIA, "NVIDIA Geforce GTX 1080," http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf, 2016, accessed on 2020-07-31.

[47] NVIDIA, "NVIDIA DGX-1: Essential Instrument for AI Research," https://www.nvidia.com/en-us/data-center/dgx-1/, 2017, accessed on 2020-07-31.

[48] NVIDIA, "NVIDIA Tesla V100 GPU Architecture Whitepaper," https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2017, accessed on 2020-07-31.

[49] NVIDIA, "NVIDIA DGX-2: The world's most powerful AI system for the most complex AI challenges," https://www.nvidia.com/en-us/data-center/dgx-2/, 2018, accessed on 2020-07-31.

[50] NVIDIA, "NVIDIA HGX-2: Powered by NVIDIA Tesla V100 GPUs and NVSwitch," https://www.nvidia.com/en-us/data-center/hgx/, 2018, accessed on 2020-07-31.

[51] NVIDIA, "NVIDIA 2019 Annual Review," https://s22.q4cdn.com/364334381/files/doc_financials/annual/2019/NVIDIA-2019-Annual-Report.pdf, 2019, accessed on 2020-07-31.

[52] NVIDIA, "NVIDIA Nsight Graphics," https://developer.nvidia.com/nsight-graphics, 2020, accessed on 2020-11-11.

[53] T. Peterka, D. Goodell, R. Ross, H.-W. Shen, and R. Thakur, "A Configurable Algorithm for Parallel Image-Compositing Applications," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. IEEE, 2009, pp. 1–10.

[54] T. Porter and T. Duff, "Compositing Digital Images," in *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*. ACM, 1984, pp. 253–259.

[55] J. W. Poulton, W. J. Dally, X. Chen, J. G. Eyles, T. H. Greer, S. G. Tell, J. M. Wilson, and C. T. Gray, "A 0.54 pJ/b 20 Gb/s Ground-Referenced Single-Ended Short-Reach Serial Link in 28 nm CMOS for Advanced Packaging Applications," *IEEE Journal of Solid-State Circuits (JSSC)*, vol. 48, no. 12, pp. 3206–3218, 2013.

[56] X. Ren, D. Lustig, E. Bolotin, A. Jaleel, O. Villa, and D. Nellans, "HMG: Extending Cache Coherence Protocols Across Modern Hierarchical Multi-GPU Systems," in *Proceedings of the 26th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 582–595.

[57] M. Segal and K. Akeley, "The OpenGL® Graphics System: A Specification (Version 4.6 (Core Profile))," https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf, 2019, accessed on 2020-07-31.

[58] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin *et al.*, "Larrabee: A Many-Core x86 Architecture for Visual Computing," *Transactions on Graphics (TOG)*, vol. 27, no. 3, pp. 1–15, 2008.

[59] Shara Tibken, "CES 2019: Moore's Law is dead, says Nvidia's CEO," https://www.cnet.com/news/moores-law-is-dead-nvidias-ceo-jensen-huang-says-at-ces-2019/, 2019, accessed on 2020-07-31.

[60] G. Stoll, M. Eldridge, D. Patterson, A. Webb, S. Berman, R. Levy, C. Caywood, M. Taveira, S. Hunt, and P. Hanrahan, "Lightning-2: A High-Performance Display Subsystem for PC Clusters," in *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*. ACM, 2001, pp. 141–148.

[61] UNIGINE, "Superposition," https://benchmark.unigine.com/superposition, 2017, accessed on 2020-11-11.

[62] M. Wimmer and P. Wonka, "Rendering Time Estimation for Real-Time Rendering," in *Eurographics Symposium onb Rendering*, 2003, pp. 118–129.

[63] C. Xie, X. Fu, and S. Song, "Perception-Oriented 3D Rendering Approximation for Modern Graphics Processors," in *Proceedings of the 24th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 362–374.

[64] C. Xie, S. L. Song, J. Wang, W. Zhang, and X. Fu, "Processing-in-Memory Enabled Graphics Processors for 3D Rendering," in *Proceedings of the 23rd International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 637–648.

[65] C. Xie, F. Xin, M. Chen, and S. L. Song, "OO-VR: NUMA Friendly Object-Oriented VR Rendering Framework for Future NUMA-Based Multi-GPU Systems," in *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*. IEEE, 2019, pp. 53–65.

[66] C. Xie, X. Zhang, A. Li, X. Fu, and S. Song, "PIM-VR: Erasing Motion Anomalies In Highly-Interactive Virtual Reality World With Customized Memory Cube," in *Proceedings of the 25th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 609–622.

[67] V. Young, A. Jaleel, E. Bolotin, E. Ebrahimi, D. Nellans, and O. Villa, "Combining HW/SW Mechanisms to Improve NUMA Performance of Multi-GPU Systems," in *Proceedings of the 51st International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 339–351.

[68] H. Yu, C. Wang, and K.-L. Ma, "Massively Parallel Volume Rendering Using 2–3 Swap Image Compositing," in *Proceedings of the conference on Supercomputing (SC)*. IEEE, 2008, pp. 1–11.