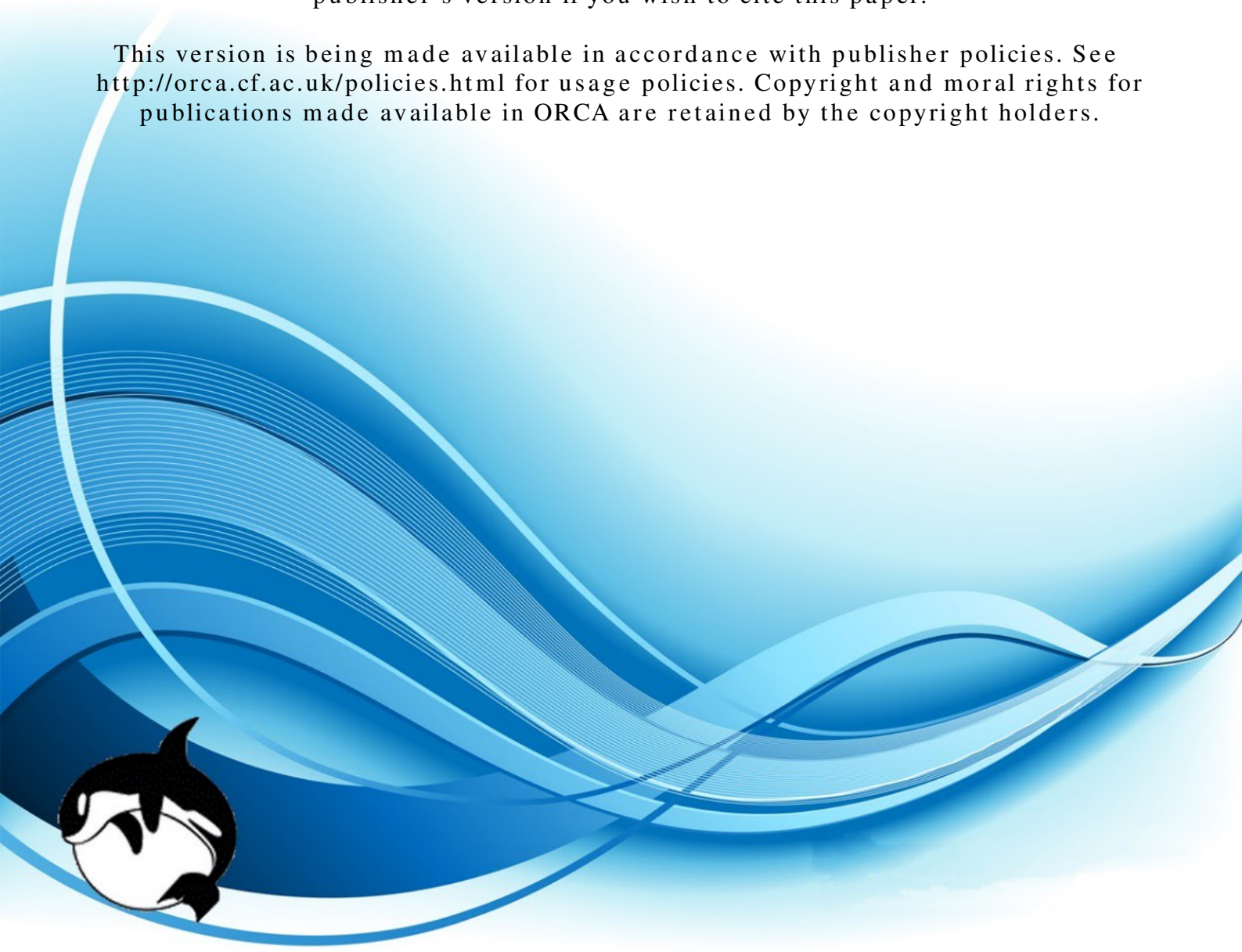

EUROGRAPHICS 2013 / I. Navazo, P. Poulin
(Guest Editors)

*Volume 31 (2013), Number 7*

# NUMA-Aware Image Compositing on Multi-GPU Platform

paperID:1084

**Abstract**

*Sort-last parallel rendering is widely used. Recent GPU developments mean that a PC equipped with multiple GPUs is a viable alternative to a high-cost supercomputer: the Fermi architecture supports uniform virtual addressing, providing a foundation for non-uniform memory access (NUMA) on multi-processor platforms. Such hardware changes require the user to reconsider the design of parallel rendering algorithms. In this paper, we propose a novel NUMA-aware image compositing algorithm, which is the key final stage in sort-last parallel rendering. Our optimal compositing algorithm is based on a proven radix-k approach, which takes advantage of NUMA architecture on the multi-GPU platform. We qualitatively analyze different image compositing approaches for practical image compositing, taking into account peer-to-peer communication costs between GPUs. Our experiments on various datasets show that our image compositing method is very fast. An image of a few megapixels can be composited in less than 10ms, which is competitive to performances reported for well-known supercomputers such as the IBM Blue Gene or Cray XT5.*

Categories and Subject Descriptors (according to ACM CCS): Haredware Architecture [I.3.1]: Parallel processing, Graphics systems [I.3.2]: distributed/network graphics, Applications [I.3.3]: Parallel renderingImage compositing

## 1. Introduction

Parallel rendering [Cro95] is an important technique for visualizing complex scenes in computer graphics, scientific visualization, CAD, and virtual reality. Parallel rendering distributes data to different processors, then sorts and composites locally-rendered data to produce the output image. According to when the sort is performed, parallel rendering can take one of three main approaches according to what is considered by each processor: sequential frames, pixels (sort-first), or graphical objects (sort-last) [MCEF94]. Unlike sequential frames and sort-first approaches, sort-last parallel rendering has the distinct advantage of high scalability and good load-balancing. Task division for parallel geometry processing and rasterization is also simple, which makes it a prime candidate to extend visualization software to high-performance parallel rendering. However, it requires the intermediate images from processing nodes to be composited to create the final image [PD84,SML*03,CMF05,PGR*09]. For an image of no more than a few tens of megapixels, this is still a very time-consuming task even for a supercomputer. For example, compositing a 64 megapixel result takes over 80ms (much longer than usable for *real-time* applications) using the IBM Blue Gene/p *Intrepid* machine at Argonne National Laboratory or the Cray XT5 *Jaguar* at Oak Ridge [KPH*10].

Multi-GPU scenarios use two or more display adapters in the same PC to speed up graphical applications via parallel processing and rendering. Multi-GPU technology has come a long way in the last few years. Single GPUs such as the NVIDIA GeForce 400 series with Fermi architecture support uniform virtual addressing [Sch11], which is the foundation of non-uniform memory access (NUMA) [SMV11] architecture on multi-processor platforms. NVIDIA's CUDA 4.0 [NVI12] contains a number of features that simplify the use of multiple GPUs within a workstation or computational node. The use of NUMA architecture [SMV11] helps to alleviate the shared memory bus bottleneck on multi-processor platforms. To take advantage of these advances, we consider *real-time* NUMA-aware image compositing using such a multi-GPU platform.

The main challenge in NUMA-aware image compositing is how to control the transfer of image data, as the communication infrastructure provided by the PCIe bus is still a limit to system performance [SMV11]. Implementing effective NUMA-aware image compositing is a non-trivial task for a number of technical reasons.

- In an ideal scenario, as the total number of processors increases in a system, the compositing throughput should scale proportionally, but this is difficult to achieve directly using CUDA 4.0.

- Various approaches to image compositing have been proposed, such as direct-send [Neu94, EP08], binary swap [lMPH94, YWM08], and recent radix-$k$ [PGR*09, KPH*10, MKPH11]. It is not obvious which is most appropriate for a multi-GPU platform.
- Times taken for reading and writing operations between peer-to-peer GPUs are different, and this must also be taken into account.

In this paper, we discuss NUMA-aware image compositing for a multi-GPU platform, and solve these technical issues in a systematic way. Our contributions include:

- An optimal NUMA-aware image compositing algorithm is proposed based on the proven radix-$k$ approach, which takes advantage of the NUMA architecture on multi-GPU platforms.
- We qualitatively analyze different image compositing approaches, taking account of peer-to-peer communication costs between GPUs.

Experiments on various datasets demonstrate that the suggested image compositing approach is very fast, and an image with a few tens of megapixels image can be composited in a PC based multi-GPU environment in under 10ms, providing a basis for real-time rendering of time-varying data (see Figure 1). Our compositing speed is competitive to the reported performance from several well-known supercomputers [KPH*10].

## 2. Related Work

### 2.1. Image Compositing

In this review, we focus on sort-last parallel rendering algorithms, as this is our chosen approach. In sort-last parallel rendering [MCEF94], object data are partitioned among $M$ processors. The objects are rendered locally, and the resulting images are depth-sorted or composited in a final step. Stompel et al. [SML*03] surveyed approaches to image compositing for this step, while Cavin et al. [CMF05] analyzed the relative theoretical performances of these methods. These overviews show image compositing algorithms can be broadly divided into one of three categories: direct-send [Neu94, EP08], binary swap [lMPH94, YWM08], and hybrid approaches.

Direct send compositing [Neu94, EP08] divides this final image gathering task into $M$ tiles to avoid exchanging full-size images. Each tile belongs to and is composited by one processor; the composited tiles are eventually assembled together to form the final image. As expected, the complexity of this algorithm is linear $O(M^2)$. The main advantage of direct send is its flexibility, since it can accommodate any number of processing nodes. It is very easy to implement and allows computation (i.e. pixel processing) to overlap with communications. However, it involves multiple processors sending messages to the same processor at the same time in an unpredictable and non-deterministic communication pattern.

The binary swap algorithm [lMPH94] is based on a binary tree compositing strategy which keeps every node busy in all stages of the process. It takes $\log_2 M$ stages to complete, where $M$ should be a power of two to fully exploit parallelism. Binary swap uses fewer messages than direct send: $O(M \log_2 M)$ messages in total, assuming minimal overlap between rounds.

To overcome the disadvantages of binary swap and direct send, many improved methods have been presented in recent years. Yu et al. proposed [YWM08] a $2-3$ swap method for parallel volume rendering which combined the advantages of both direct send and binary swap. It avoids all-to-all communication and considers binary swap as a special case of a $2-3$ swap algorithm.

Radix-$k$ compositing [PGR*09, KPH*10, MKPH11] was later introduced a a configurable parallel image compositing method. The radices are a set of configurable parameters represented as a vector $\mathbf{k} = [k_1, k_2, ...k_r]$, where $M = \prod k_i$, and $r$ denotes the number of communication and compositing rounds. During each round $i$, the $M$ processing nodes are divided into $M/k_i$ groups of $k_i$ participants, which communicate only within their group in a direct send fashion. When all $k$-values are equal to 2, this is equivalent to the binary swap algorithm. When there is a single $k$-value equal to $M$, there is only a single round and this is equivalent to the direct send algorithm. Radix-$k$ compositing has been shown to perform better than the binary swap and $2-3$ swap algorithms. However, the biggest obstacle to applying the radix-$k$ algorithm is the lack of a clear strategy for determining the vector $\mathbf{k}$ which provides optimal performance, since the choice of radices depends on network topology and hardware. Attempts have been made [PGR*09, KPH*10] to find the best radices for particular hardware platforms by executing a series of benchmarks, rather than giving an analysis. Here, we consider the optimal radix$-k$ approach from a theoretical viewpoint by considering the properties of multi-GPU PC systems; we later validate our claims.

### 2.2. Multi-GPU Systems

To simplify programming of rendering on multi-GPU systems, Moerschell and Owens [MO06] presented a consistent, distributed, shared memory system. Recent parallel rendering researching mainly focuses on advances in multi-GPU clusters, such as ones based on *InfiniBand* fat-trees [CD12]. GPUs attached to different I/O ports can not access each other on a peer-to-peer (P2P) basis, so this does not provide a true NUMA architecture. NVIDIA developed CUDA 4.0 [NVI12] to support multiple GPU parallel rendering using NUMA architecture on a PC. However, taking full advantage of it requires careful analysis and thought..

Stephane et al. [MMD08] implemented sort-last volume

**Figure 1:** *Real-time rendering of time-varying data. A 16 megapixel resolution image is rendered on a 4-GPU platform in about 5ms.*

visualization on a multi-GPU system, and analyzed bottlenecks in and advantages of multi-GPU systems. Spafford et al. [SMV11] quantitatively analyzed the benefits of NUMA architecture on a multi-GPU platform, and provided guidance on programming strategies to maximize performance. The NUMA architecture was initially analyzed by [EBA*12] for hybrid multi-GPU clusters to optimize asynchronous parallelization of rendering stages. In our case, we specifically consider the problem of image compositing on a NUMA multi-GPU platform.

## 3. Background

### 3.1. Multi-GPU NUMA Architecture

As illustrated in Figure 2, $M$-GPUs in NUMA architecture are connected in a binary tree structure. The GPUs are at the leaves of a full binary tree, and internal tree nodes are PCIe switches except for the root node which is an I/O hub (IOH). Note that $M$ need not necessarily be a power of two, and certain leaves could be empty. Although all GPUs can access each other directly, the PCIe bandwidth ($B$) is much narrower than that of video memory. The performance of P2P access between GPUs is still bounded by the limited bandwidth of each PCIe switch (denoted as $B_s$), which causes the bottleneck in image compositing. As noted by NVIDIA nd confirmed in practice, the highest P2P communication speed is achieved between GPUs on the same PCIe switch. Such a communication path does not include the IOH; the bandwith of the IOH (denoted as $B_h$) is lower.

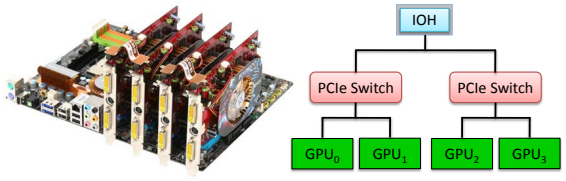


**Figure 2:** *Example of multi-GPU NUMA architecture (right) on a 4-GPU platform (left). GPUs are connected by PCIe switches and an I/O hub (IOH) in a tree structure.*

Image compositing must communicate image data using reading and writing operations. Writing speed is noted by NVIDIA to be lower than reading during P2P GPU access, and bidirectional throughput is usually less than twice that of unidirectional transfers: the PCIe bus between GPUs is not an ideal full duplex connection.

Some concepts relevant to NUMA architecture are introduced here for further usage:

- **Local GPU**: the GPU to which the compositing result image belongs;
- **Neighboring GPU**: a GPU which can be accessed by P2P by the local GPU;
- **Remote GPU**: a GPU which cannot be accessed by P2P by the local GPU.
- **GPU-pair**: two GPUs connected by PCIe switches and/or the IOH.

### 3.2. NUMA-aware compositing algorithm overview

We study how the radix-$k$ algorithm can be used for greatest compositing speed. The keys to improving radix-$k$ performance are: increasing message concurrency, avoiding contention, and overlapping communication with computation.

As video memory throughput may be 20 or more times greater than PCIe throughput, we assume we can ignore the cost of local video memory accesses (due to computation) in many cases, and focus on quantifying the effect of P2P accesses (needed for communication) on image compositing.

PCIe link contention always occurs due to the complex communication between GPUs in NUMA architecture. As Figure 2 shows, GPUs are interconnected by PCIe switches and the IOH, and data streams scatter or gather at these interconnected nodes. The PCIe bus is a duplex connection, and can communicate concurrently in both directions. However, when more than one data stream path overlaps in the same direction, PCIe link contention will happen, as illustrated by examples in Figure 3. Quantitatively approaching the link contention is critical to analyzing the communication cost.

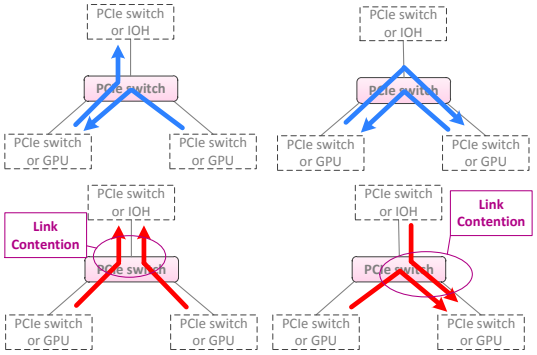


**Figure 3:** *Link contention at PCIe switches occurs if data flows overlap in the same direction. Top: no contention. Bottom: contention occurs.*

## 4. Multi-GPU NUMA-aware Image Compositing

In the following section, we first find the optimal **k**-vector for the radix-$k$ method on NUMA architecture (Section 4.1), then we address how to build practical GPU-pair compositing approaches by considering the reading and writing communication costs (see Section 4.2).

### 4.1. Optimal Compositing on the Multi-GPU Platform

Enlightened by the approach of mathematical analysis of collective communication [CHPvdG07], we define the image compositing cost function as follows:

$$T = f(\mathbf{k}, N, \mathbf{B}), \tag{1}$$

where **k** is the **k**-vector in the radix-$k$ algorithm, $N$ is the number of pixels in each partial image which participates in composition, and **B** is a bandwidth vector related to the grouping approaches as explained later.

Given a compositing strategy specified by a vector **k**, Function 1 returns the minimal compositing time using this strategy. For example, $f([8], N, [L_3])$ is the minimal compositing time using the direct send algorithm for 8 GPUs, while $f([2,2,2], N, [L_1, L_2, L_3])$ is the time taken for the binary swap approach. For any given vector $\mathbf{k} = [k_1, k_2 ... k_t]$ where $M = \prod_{i=1}^{t} k_i$ and $M$ is the number of GPUs, analysis of the radix-$k$ procedure [PGR*09] shows that the following equation holds:

$$f([k_1, ...k_t], N, [B_{k_1}, ...B_{k_t}]) = \sum_{i=1}^{t} f([k_i], \frac{N}{\prod_{j=0}^{i-1} k_j}, [B_{k_i}]). \tag{2}$$

Here, we additionally define $k_0 = 1$ to unify the expression of the cost function. In Equation 2, each item of the sum corresponds to the cost of one compositing round in the algorithm, and the total composition time is the sum of these times.

For NUMA architecture structured in a full binary tree, we claim that the optimal radix-$k$ compositing strategy is the binary swap method. We prove this claim using the above formulation.

**Theorem 1** Suppose $k_i$ GPUs are grouped for compositing, where $k_i$ is a power of two. Let $N$ be the number of pixels in the current image to be composited in this round. Then if $k_i \geq 4$, the following holds:

$$f([k_i], N, [B_{k_i}]) \geq f([\frac{k_i}{2}, 2], N, [B_{k_{i-1}}, B_{k_i}]) \tag{3}$$

*Proof* As all GPUs are linked in a binary tree, a PCIe switch (or the IOH) must be the lowest common ancestor (nearest the root) for the current $k_i$ GPUs. We call this ancestor $A_{k_i}$ and suppose its pixel bandwidth is $B_{k_i}$ as defined previously. The cost of $f([k_i], N, [B_{k_i}])$ is just the cost of direct send for $k_i$ GPUs; each GPU is in charge of compositing $N/k_i$ pixels. As an intrinsic property of direct send, the number of pixels passing through $A_{k_i}$ unidirectionally is $(k_i/2)^2 \cdot N/k_i$, so the following inequality holds:

$$f([k_i], N, [B_{k_i}]) \geq (\frac{k_i}{2})^2 \cdot \frac{N}{k_i} \cdot \frac{1}{B_{k_i}} \tag{4}$$

In the worst case, each compositing step of a single GPU encounters link contention. This gives an upper bound to the direct send cost as follows (here we let $r = \log_2 k_i - 1$):

$$f([k_i], N, [B_{k_i}]) \leq \sum_{j=0}^{r} (2^j)^2 \cdot \frac{N}{k_i} \cdot \frac{1}{B_{k_i}} \leq \frac{k_i^2 - 1}{3} \cdot \frac{N}{k_i} \cdot \frac{1}{B_{k_i}} \tag{5}$$

Next we find bounds on $f([k_i/2, 2], N, [B_{k_{i-1}}, B_{k_i}])$. The compositing procedure for the $k$-vector $[k_i/2, 2]$ has two steps. Firstly $k_i/2$ GPUs are grouped, and direct send is executed to composite images, then between-group GPUs are paired to complete the final composition. The time for the first step can be recursively represented as $f([k_i/2], N, [B_{k_{i-1}}])$, but the time for the second step needs further consideration. After finishing the first compositing step, just $k_i/2$ pixels pass through the root $A_{k_i}$ in the second step. Thus, the compositing time for the second step is $(N/2) \cdot (1/B_{k_i})$. So the following equation holds:

$$f([\frac{k_i}{2}, 2], N, [B_{k_{i-1}}, B_{k_i}]) = f([\frac{k_i}{2}], N, [B_{k_{i-1}}]) + \frac{N}{2B_{k_i}} \tag{6}$$

Using Equation 5 and $B_{k_{i-1}} \geq B_{k_i}$, the upper bound of $f([\frac{k_i}{2}, 2], N, [B_{k_{i-1}}, B_{k_i}])$ is:

$$f([k_i/2, 2], N, [B_{k_{i-1}}, B_{k_i}]) \leq \frac{k_i^2 - 4}{6} \cdot \frac{N}{k_i} \cdot \frac{1}{B_{k_i}} + \frac{N}{2} \cdot \frac{1}{B_{k_i}} \tag{7}$$

A simple function $G(k_i, N)$ can be constructed as follows:

$$G(k_i, N) = f([k_i], N, [B_{k_i}]) - f([\frac{k_i}{2}, 2], N, [B_{k_{i-1}}, B_{k_i}]). \tag{8}$$

After substituting Inequalities 4 and 7, we can easily conclude that $G(k_i, N) \geq 0$ when $k_i \geq 4$, proving Theorem 1. □

In practice, it is extremely unlikely that equality is reached in Inequalities 4 and 5, so we may assume that $f([k_i],N,[B_{k_i}])$ is always greater than $f([k_i/2,2,[B_{k_{i-1}},B_{k_i}]],N)$. Thus, given any radix-$k$ vector $\mathbf{k}=[k_1,k_2,...k_t]$, we can find the minimal compositing time by recursively using Theorem 1 to decompose the vector $\mathbf{k}$ for any element $k_i \geq 4$. Eventually, all elements of $\mathbf{k}$ must be two, and this is exactly the $k$-value for binary swap. One more key point to note is that the grouping order of binary swap must be bottom-up, that is, each GPU should firstly swap pixels with its nearest partner. This grouping order will minimize the number of pixels crossing PCIe switches, and maxmize the performance of binary swap. In summary, we arrive at the important conclusion that binary swap is the optimal image compositing method for a NUMA multi-GPU platform.

### 4.2. GPU-pair Image Compositing Modes

Although we have found the theoretically optimal compositing strategy, we still need to consider many factors (e.g. PCIe communications) to arrive at the best algorithm in practical terms. In this section we consider various GPU-pair compositing approaches, and quantitatively consider how NUMA architecture affects image compositing.

For a GPU-pair $GPU_0$ and $GPU_1$, $GPU_0$ communicate with $GPU_1$ through one or more PCIe switches (or the IOH), and they can read or write each other's GPU memory. By removing obviously low performance cases, we arrive at three different compositing approaches as shown in Figure 4. On the left of Figure 4, each GPU in a pair mutually reads the color (C) and depth (D) images from its neighbor to local memory, and composites partial images locally: we call this case the *mutual read compositing* (MRC) approach. In the middle case, each GPU in a pair mutually writes color and depth images from local memory to their neighbors, then images are composited on the local GPU: we call this *mutual write compositing* (MWC). The right case is more complicated and involves hybrid inter-GPU communication. Firstly, each GPU writes depth information to the neighboring GPU's memory. Secondly, both GPUs read color images from their neighbors based on the depth image read in the first step: we call this approach *mutual write-read compositing* (MWRC).

The compositing process involves many reading and writing operations between GPUs, and it is not straightforward to determine which approach is best. Performance variations occur between between P2P reading and writing, and these directly affect compositing performance. Suppose the time for a GPU to read one pixel from another is $\alpha$, and the time for writing is $\beta$. To describe the imbalance between reading and writing, we define the coefficient $\varepsilon$ to be:

$$\beta = \varepsilon\alpha. \tag{9}$$

For a given multi-GPU system, $\varepsilon$ is a constant parameter determined by the PCIe bus and system chipset.

In addition to differences in reading and writing, another factor to consider is that PCIe is not an ideal full duplex bus. As noted by the manufacturer, and observed in practice, the bidirectional bandwidth of PCIe is less than twice its unidirectional bandwidth; further investigation also shows that there are performance variations between duplex P2P reading and writing. We define two further ratios as below to relate simplex and duplex P2P accesses:

$$k_\alpha = T_{DR}/T_{SR}, \tag{10}$$
$$k_\beta = T_{DW}/T_{SW}, \tag{11}$$

where $T_{DR}$ (or $T_{DW}$) is the time taken by a GPU to read (or write) image data from its neighboring GPU in bidirectional (duplex) operation, and $T_{SR}$ (or $T_{SW}$) is the time for reading (or writing) $n$ data items in unidirectional (simplex) operation. $k_\alpha$ and $k_\beta$ thus describe the decrease in PCIe bandwidth when using duplex, relative to simplex. If duplex throughput is exactly twice that of simplex, then $k_\alpha$=1 and $k_\beta$=1, but $k_\alpha$ and $k_\beta$ are usually greater than 1, since duplex throughput is less than twice of simplex.

To quantitatively express the cost of these three compositing approaches, we show what happens over time in each in Figure 5. Assume that $R_0$ and $R_1$ ($R_0 \geq R_1$) are the resolutions of images $I_0$ and $I_1$ in $GPU_0$ and $GPU_1$, $I_0$ and $I_1$ are separately composed of $I_{00}$ and $I_{01}$, and $I_{10}$ and $I_{10}$ respectively, and the number of active pixels in $I_0$ and $I_1$ is $R$. We can on average distribute the active pixels of $I_0$ (or $I_1$) into $I_{00}$ (or $I_{10}$) and $I_{01}$ (or $I_{11}$) by equally separating pixels. Then the cost time of the three approaches is:

$$T_{MRC} = \alpha k_\alpha \frac{R+R_1}{2} + \alpha\frac{R_0-R_1}{2} \tag{12}$$
$$T_{MWC} = \beta k_\beta R_1 + \beta(R_0-R_1) \tag{13}$$
$$T_{MWRC} = \beta(k_\beta\frac{R_1}{2} + \frac{R_0-R_1}{2}) + \alpha(k_\alpha\frac{R_1}{2} + \frac{R_0-R_1}{2}) \tag{14}$$

All three cost expressions are the sum of two terms. The first term in Equations 12 and 13 represents the cost of bidirectional reading or writing. The second term is the time for unidirectionally transferring $\frac{R_0-R_1}{2}$ pixels. Equation 14 is a hybrid of Equations 12 and 13 as it mixes reading and writing operations.

A key point concerns which pixels are active. In MRC, we have to read full resolution partial images since there is no information about active pixels. MWC and MWRC differ because the local GPU knows which pixels are active and can therefore avoid writing blank pixels. This is why the term $C$ appears in Equation 12 but not in Equations 13 and 14. On the other hand, if we have already read depth images, only active pixels need be transferred when reading color images.

We have discussed the case of $Z$-buffer composition

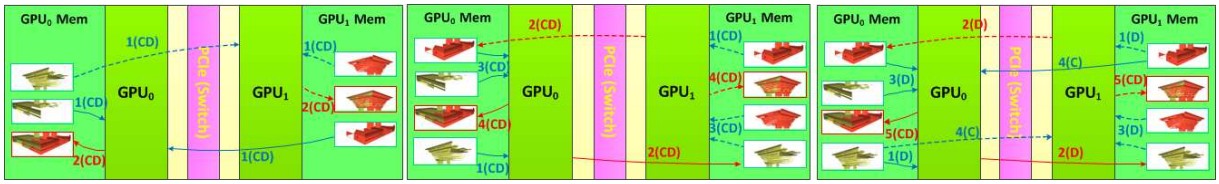


**Figure 4:** *Compositing approaches for a GPU-pair: left to right: MRC, MWC and MWRC. Red lines indicate writing operations; blue lines indicate reading. Arrows indicate direction of data transfer. C and D represent color and depth images respectively.*

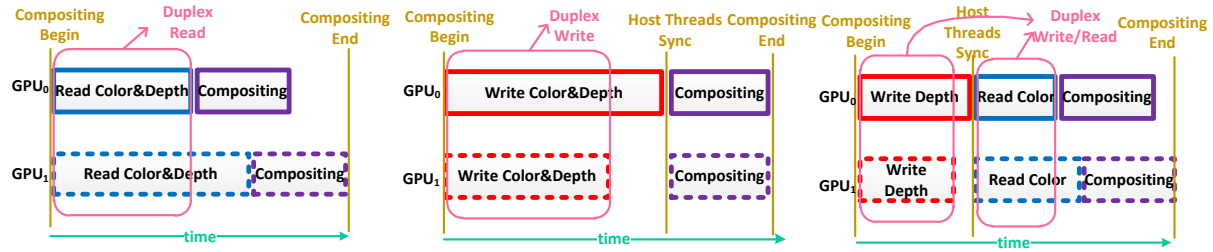


**Figure 5:** *Sequence diagram for each compositing approach. Left to right: MRC, MWC and MWRC. Synchronization points are needed for MWC and MWRC, as they include read-after-write dependencies.*

above. But most parallel volume rendering composites images by a sorting processes, which does not need a $Z$-buffer. If we directly composite images without depths, the compositing time expressions become:

$$T_{MRC-NZ} = \alpha k_\alpha \frac{R}{2} \quad (15)$$

$$T_{MWC-NZ} = \beta k_\beta \frac{R_1}{2} + \beta \frac{R_0 - R_1}{2} \quad (16)$$

We can ignore the mutual write-read method in this case because of the absence of depth information. Fewer pixels are transferred between GPUs.

Since $k_\alpha$, $k_\beta$, $\alpha$ and $\beta$ are all hardware platform dependent, we can draw the conclusion that $T_{MRC}$, $T_{MWC}$, $T_{MWRC}$, $T_{MRC-NZ}$, $T_{MWC-NZ}$ are all determined by the input active pixels in a given system. Specifying two input images, we can evaluate the above compositing time easily, and decide which composition approach should be used.

## 5. Experimental Results

During the experiments, we mainly considered two issues. Firstly, we measured the parameters of our multi-GPU platforms since they are crucial in analysing compositing performance. Secondly, the performance of different compositing approaches was experimentally determined, for various **k**-vectors, to verify our analysis and expectations.

**Table 1:** *Two Multi-GPU platforms configurations*

| | Gigabyte UD9 | ASUS ESC4000 |
|---|---|---|
| CPU | Intel i7 950 2.8GHz | Intel Xeon X5675 3.07Hz |
| Memory | 12GB | 48GB |
| GPU | GeForce GTX 480 | GeForce GTX 460 |
| PCIe | PCIe 2.0 X16 | PCIe 2.0 X8 |
| Motherboard | X58A-UD9 | Intel (R) 5520 |

### 5.1. Test Environment

We used two different platforms to test our NUMA-aware image compositing ideas. One was a PC and the other was a server node, equipped with 4 and 8 GPUs, respectively in configurations shown in Table 1. For simplicity, we call the 4 GPU system UD9 and the 8 GPU platform ESC4000. Although the ESC4000 was equipped with 8 PCIe slots, their bandwidth was only X8 as all PCIe slots were filled. However, this degradation of bandwidth did not effect the scalability performance testing. The GPU architectures of the two platforms are shown in Figure 6.

### 5.2. Determining Parameters for a GPU-Pair

To determine the differences in P2P reading and writing, we used batched reading and writing to obtain mean performance for each GPU-pair. On each platform, we used different size images as test data, and computed mean access bandwidth for simplex reading (SR), simplex writing

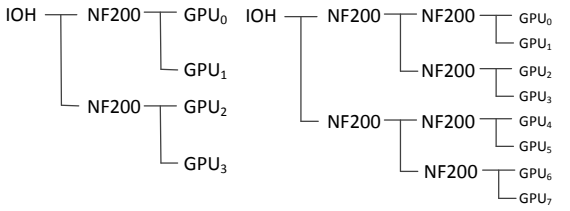


**Figure 6:** *Two platform illustrations: Left: UD9; Right ESC4000.*

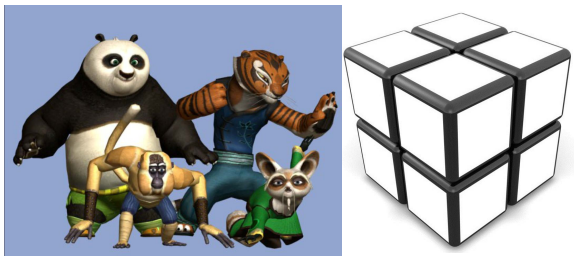

**Figure 7:** *16-megapixel image compositing on UD9 (left) and ESC400 (right) in under 10ms.*

(SW), duplex reading (DR) and duplex writing (DW). As mentioned above, there are two types of bandwidth to be determined: one is the bandwidth of paths which only include PCIe switches, the other is for paths including the IOH. For example, the path between $GPU_0$ and $GPU_1$ on both UD9 and ESC4000 does not include the IOH, while the GPU-pair $GPU_{(0,2)}$ on UD9 and the pair $GPU_{(0,4)}$ on ESC4000 include the IOH. Thus, we separately list their P2P bandwidth performance in Table 2.

**Table 2:** *P2P bandwidth of our NUMA platforms*

| | Reading/Writing Bandwidth(unit: GB/s) | | | | |
|---|---|---|---|---|---|
| Platform | GPU-pair | SR | SW | DR | DW |
| UD9 | $GPU_{(0,1)}$ | 6.17 | 5.22 | 10.0 | 10.40 |
| | $GPU_{(0,2)}$ | 4.44 | 3.29 | 5.98 | 6.5 |
| ESC4000 | $GPU_{(0,1)}$ | 3.09 | 2.97 | 5.46 | 5.96 |
| | $GPU_{(0,2)}$ | 2.54 | 2.46 | 4.66 | 5.02 |
| | $GPU_{(0,4)}$ | 2.54 | 2.45 | 4.65 | 5.00 |

**Table 3:** *Parameters of our NUMA platforms*

| GPU-pair | $GPU_{(0,1)}$ | | | $GPU_{(0,2)}$ | | |
|---|---|---|---|---|---|---|
| | $\varepsilon$ | $k_\alpha$ | $k_\beta$ | $\varepsilon$ | $k_\alpha$ | $k_\beta$ |
| UD9 | 1.18 | 1.23 | 1.0 | 1.35 | 1.48 | 1.01 |
| | | | | | | |
| GPU-pair | $GPU_{(0,1)}$ | | | $GPU_{(0,4)}$ | | |
| | $\varepsilon$ | $k_\alpha$ | $k_\beta$ | $\varepsilon$ | $k_\alpha$ | $k_\beta$ |
| ESC4000 | 1.04 | 1.13 | 1.0 | 1.03 | 1.09 | 0.98 |

The values for, $\varepsilon$, $k_\alpha$, $k_\beta$ are constant for various applications, and can help to determine how to achieve optimal performance. For example, on the UD9 platform, $\varepsilon$ of $GPU_{(0,1)}$ is about 1.18, i.e. the throughput of simplex P2P reading is 1.18 times than writing. $k_\alpha = 1.48$ for $GPU_{(0,2)}$ indicates that the performance of reading from $GPU_0$ to $GPU_2$ is a factor 1.48 times lower in the duplex cases than in the simplex case. On the ESC4000 system, all three parameters for both GPU-pairs are closer to 1.0 than for UD9: the NUMA architecture has less impact here. $k_\beta$ for both systems is close to 1.0 for any GPU-pair, meaning that there is little effect when switching from simplex writing to duplex.

### 5.3. Image Compositing Performance Tests

We tested the image compositing performance on UD9 and ESC400 by using triangle meshes with over $10^9$ triangles (see Figure 1 and 7). The final image size was set to 4K $\times$ 4K, providing a rendering speed under 10ms for the three image compositing approaches discussed. Such allows real-time rendering of time-varying mesh sequences as in Figure 1. We selected UD9 as our experimental platform for further study of how the NUMA architecture impacts image compositing (page limits preclude reporting on ESC400 too). We chose UD9 as it had greater performance variances than ESC4000 so illustrates our analysis more clearly.

Given fixed $k_\alpha$, $k_\beta$, $\varepsilon$ and image resolution, the compositing time is determined by the number of active pixels in each of the two input images. Thus our test used images with vary numbers of active pixels to examine the validity of our approach proposed in Section 4.2.

To obtain different active pixels with fixed resolution, we used two different camera views of the mesh. We refer to the first scene as Scale-1, and the second as Scale-2. In Scale-1 (or Scale-2), the ratio of active pixels for $GPU_0$, $GPU_1$, $GPU_2$, and $GPU_3$ were respectively 10% (or 18%), 15% (or 30%), 17% (or 33%), and 24% (or 43%).

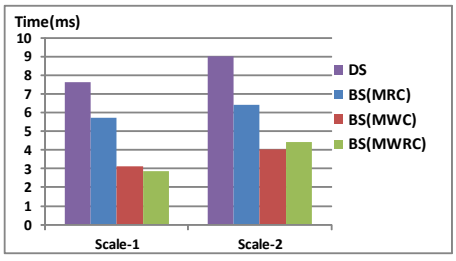


**Figure 8:** *Times for direct send (DS) and three implementations of binary swap (BS) on UD9.*

To illustrate the variations in the three compositing approaches, we implemented binary swap in different ways as described earlier on UD9. The compositing times using four GPUs on UD9 are shown in Figure 8. All three implementations of binary swap have better performance than direct send. When the partial image of each GPU is sparse, as in Scale-1, the time taken by MRC is almost twice that of MWRC and MWC. This is because the costs of MWRC and MWC only depend on active pixels. while MRC depends on the full resolution of the input image, not just active pixels.

With a greater proportion of active pixels in partial images (going from Scale-1 to Scale-2), the compositing times for MWRC and MWC obviously increases, as shown in Figure 8; the growth rates for MWRC and MWC are higher than for direct send and MRC. This accords with the prediction of our cost approach. Another detail we observed is that MWRC takes less time than MWC for Scale-1, but longer than MWC for Scale-2. This occurs because $k_\alpha$ is much larger than $k_\beta$ on UD9, and the partial images in Scale-1 are sparser than in Scale-2. The slow duplex reading operation increases the costs of MWRC when there are more active pixels to composite.

### 5.4. Image Compositing Scalability Tests

To test the scalability of our compositing approach, we chose a large volume data set from a practical application and visualized it by ray casting. The volume data was the electromagnetic field generated by a particle in cell (PIC) simulation, with dimensions $720 \times 720 \times 960$. Each grid point is a single-precision floating-point number, so the total size of this dataset is about 1.8 GB. We used the binary swap by MWC method in this test, as it only depends on active pixels.

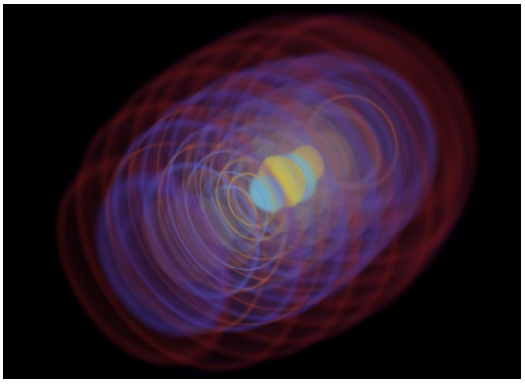

**Figure 9:** *Volumetric electromagnetic field with* 1.8*GB, rendered by ESC4000.*

Before ray casting the volume data, we divided the dataset using a $k$-D-tree strategy, and statically distributed sub data blocks onto GPUs. The number of $k$-D-tree leaf nodes was equal to the number of GPUs. Unlike polygon mesh rendering, we composited partial images by sorting the ray casting results, and used OVER / UNDER operators to blend each pixel using the GPU. Thus, the communication cost for this test follows Equation 16—there is no depth information transferred in image composition. The aim of this experiment was to first verify the optimality of binary swap, as demonstrated in Section 4.1, and then to analyze the scalability of parallel rendering on ESC4000.

There are four different compositing strategies for 8 GPUs, and each strategy exactly corresponds to one **k**-vector in the radix-$k$ algorithm. Figure 10 illustrates all four **k**-vectors used in this test and their compositing times. The $k$-vector [2,2,2] corresponds to binary swap, while the **k**-vector [8] corresponds to direct send. The whole composition is divided into the same number of stages as the length of the **k**-vector. For example, the length of [2,2,2] is three, so there are three stages in binary swap, and so there are three synchronization points in the composition procedure.

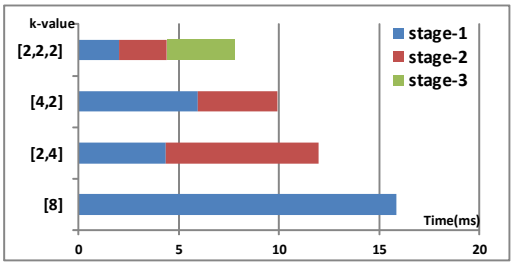


**Figure 10:** *Different radix-k image compositing times on ESC400.*

As shown in Figure 10, binary swap is much faster than direct send, and the other two methods ([4,2] and [4,2]) lie in between. Approach [4,2] is better than [2,4] because there are fewer pixels passing through the IOH. These experimental results confirm our analysis, and we can say with confidence that binary swap is the optimal compositing method for multi-GPU systems.

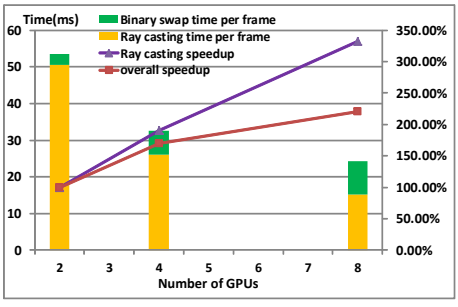


**Figure 11:** *scalability test of different GPUs on ESC4000.*

To measure the overall performance of parallel rendering, we did a scalability test on ESC4000. As binary swap needs a power-of-two threads, only cases with 2, 4 and 8 GPUs were measured, as shown in Figure 11. The overall rendering time has two parts: ray casting time and binary swap time. The test volume data was nearly 2 GB, and cannot be entirely loaded into a single GPU memory. Thus, we took the the case of 2 threads as the baseline, and calculated speedup relative to this case. With two threads, it only took a short time to composite the two partial images. With increasing numbers of threads, the ray casting time became shorter, while the compositing time increased gradually.

If we ignore the cost of composition, the scalability of ray casting is high (indicated by the purple line in Figure 11). The frame rate with 8 GPUs rendering is about 3.3 times the rate for 2 GPUs. However, considering the compositing cost, the overall speedup is lower (indicated by the red line in Figure 11). The cost of composition becomes the bottleneck with increasing numbers of rendering threads. As the PCIe bandwidth of ESC4000 is X8, this narrow throughput hampered the benefits of parallelism.

## 6. Conclusions and Future Work

We have considered image compositing on NUMA architecture multi-GPU systems. In this case, we have proved that binary swap is the best NUMA-aware image compositing strategy among all radix-*k* methods. To make best use of P2P communication, three GPU-pair compositing approaches were considered, and we quantitatively evaluated the mathematical relationship between compositing time and the number of active pixels. Two hardware platforms were used to confirm the performance differences of P2P reading and writing, and all test results on various data support our analysis.

This research offers some avenues for further exploration. For example, the NUMA-aware cost approach could also provide a theoretical basis for further optimizing compositing algorithms, potentially leading to more efficient compositing strategies.

## References


[CD12] CAVIN X., DEMENGEON O.: Shift-Based Parallel Image Compositing on *InfiniBand*$^{TM}$ Fat-Trees. pp. 129–138. 2

[CHPvdG07] CHAN E., HEIMLICH M., PURKAYASTHA A., VAN DE GEIJN R.: Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience 19*, 13 (2007), 1749–1783. 4

[CMF05] CAVIN X., MION C., FILBOIS A.: Cots cluster-based sort-last rendering: Performance evaluation and pipelined implementation. In *IEEE Visualization* (2005), IEEE Computer Society, p. 15. 1, 2

[Cro95] CROCKETT T. W.: *Parallel Rendering*. Tech. rep., 1995. 1

[EBA*12] EILEMANN S., BILGILI A., ABDELLAH M., HERNANDO J., MAKHINYA M., PAJAROLA R., SCHÜRMANN F.: Parallel rendering on hybrid multi-gpu clusters. In *Eurographics Symposium on Parallel Graphics and Visualization* (2012), pp. 109–117. 3

[EP08] EILEMANN S., PAJAROLA R.: Direct send compositing for parallel sort-last rendering. In *ACM SIGGRAPH ASIA 2008 courses* (New York, NY, USA, 2008), ACM, pp. 39:1–39:8. 2

[KPH*10] KENDALL W., PETERKA T., HUANG J., SHEN H.-W., ROSS R. B.: Accelerating and benchmarking radix-k image compositing at large scale. In *Eurographics Symposium on Parallel Graphics and Visualization* (2010), pp. 101–110. 1, 2

[lMPH94] LIU MA K., PAINTER J. S., HANSEN C. D.: Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Applications 14* (1994), 59–68. 2

[MCEF94] MOLNAR S., COX M., ELLSWORTH D., FUCHS H.: A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications 14* (1994), 23–32. 1, 2

[MKPH11] MORELAND K., KENDALL W., PETERKA T., HUANG J.: An image compositing solution at scale. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2011), ACM, pp. 25:1–25:10. 2

[MMD08] MARCHESIN S., MONGENET C., DISCHLER J.-M.: Multi-gpu sort-last volume visualization. In *Eurographics Symposium on Parallel Graphics and Visualization* (2008), Eurographics Association, pp. 1–8. 2

[MO06] MOERSCHELL A., OWENS J. D.: Distributed texture memory in a multi-gpu environment. In *Graphics Hardware* (Sept. 2006), Olano M., Slusallek P., (Eds.), pp. 31–38. 2

[Neu94] NEUMANN U.: Communication costs for parallel volume-rendering algorithms. *IEEE Computer Graphics and Applications 14*, 4 (1994), 49–58. 2

[NVI12] NVIDIA: Cuda toolkit 4.0, 2012. http://developer.nvidia.com/cuda-toolkit-40. 1, 2

[PD84] PORTER T., DUFF T.: Compositing digital images. In *SIGGRAPH* (1984), pp. 253–259. 1

[PGR*09] PETERKA T., GOODELL D., ROSS R., SHEN H.-W., THAKUR R.: A configurable algorithm for parallel image-compositing applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (New York, NY, USA, 2009), ACM, pp. 4:1–4:10. 1, 2, 4

[Sch11] SCHROEDER T. C.: *Peer-to-Peer and Unified Virtual Addressing*. Tech. rep., 2011. 1

[SML*03] STOMPEL A., MA K.-L., LUM E. B., AHRENS J., PATCHETT J.: Slic: Scheduled linear image compositing for parallel volume rendering. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (Washington, DC, USA, 2003), IEEE Computer Society, pp. 6–12. 1, 2

[SMV11] SPAFFORD K., MEREDITH J. S., VETTER J. S.: Quantifying numa and contention effects in multi-gpu systems. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units* (New York, NY, USA, 2011), GPGPU-4, ACM, pp. 11:1–11:7. 1, 3

[YWM08] YU H., WANG C., MA K.-L.: Massively parallel volume rendering using 2–3 swap image compositing. In *ACM SIGGRAPH ASIA 2008 courses* (New York, NY, USA, 2008), ACM, pp. 40:1–40:11. 2