Timeline Scheduling for Out-of-Core Ray Batching: Supplementary Report

Myungbae Son
KAIST
nedsociety@kaist.ac.kr

Sung-Eui Yoon
KAIST
sungeui@kaist.edu

ACM Reference format:
Myungbae Son and Sung-Eui Yoon. 2017. Timeline Scheduling for Out-of-Core Ray Batching: Supplementary Report. In *Proceedings of HPG '17, Los Angeles, CA, USA, July 28-30, 2017,* 4 pages.
DOI: 10.1145/3105762.3105784

Treating data transfers as separate jobs allocated to channels. We treat a data transfer as a separate job and let our scheduler to allocate those jobs to memory channels. This abstraction is reasonable for data transfers between main memory devices and between a disk and main memory, thanks to the wide adoption of direct memory access (DMA), enabling asynchronous access [Osborne 1980]. For the data transfer between a CPU and a GPU, an asynchronous full-duplex data transfer might not overlap with kernel execution based on the internal engine implementation [Harris 2012]. In this case, we simply create a dummy job to the GPU side, to mark it busy due to processing the data transfer.

Detailed job structures. When a client requests rendering a frame (**ImageRequest**) with a camera parameters (`CameraInfo`), it sets up an empty `Framebuffer` for each device and the initial jobs (**RayIntersect**) processing sets of primary rays (`RayQueue`). From there, each secondary and shadow rays are queue up to `(Shadow)RayQueue`. Each ray in `(Shadow)RayQueue` holds the reference to the `Framebuffer` and the sample index, i.e. pixel location, to shade. Each queue is associated with a `SceneData` block, defined by a scene subdivision (e.g., uniform grid or kd-tree). They are then traced in **(Shadow)RayIntersect** jobs. Finally, when all samples are processed, the results are gathered back to the client in an **AdditiveCompose** job, where a set of partially shaded `Framebuffers` are summed up to the result image.

Effects of job prediction. Our prediction method does not consider actual geometry, but considers the average behavior given a ray queue for fast prediction performance. To see its effects, we have measured the miss rate of our prediction method, and its average improvement over the other case that does not perform any prediction.

Specifically, we analyzed two types of prediction misses. First, for two or more jobs with the same fetching overhead, it is possible that the prediction results in incorrect number of rays, causing a job with lower granularity (that is, a job with a smaller number of rays) to be preferred over the jobs with a higher granularity, which causes *mis-ordering* of job allocation. It is measured by the ratio of wrong choices on the predicted E-job selection. Another case is that when rays are distributed in non-uniform ways, the scheduler predicts imaginary ray queues to be generated, causing unnecessary T-jobs to be scheduled, resulting in *mis-fetching*. It is measured by the ratio of T-jobs for predicted E-jobs that are non-existent.

	Boeing777	***SponzaMuseum***
Mis-ordering	11.4 %	10.8 %
Mis-fetching	21.5 %	13.1 %
Total prediction miss rate	32.9 %	24.0 %
Avg. improvement	**68.1 %**	**84.5 %**

Table 2: This table shows miss rates of our prediction method and the average improvement of our method over the case running w/o any prediction. This data is computed by averaging inspections per scene over 50 runs in the homogeneous 4-node setup.

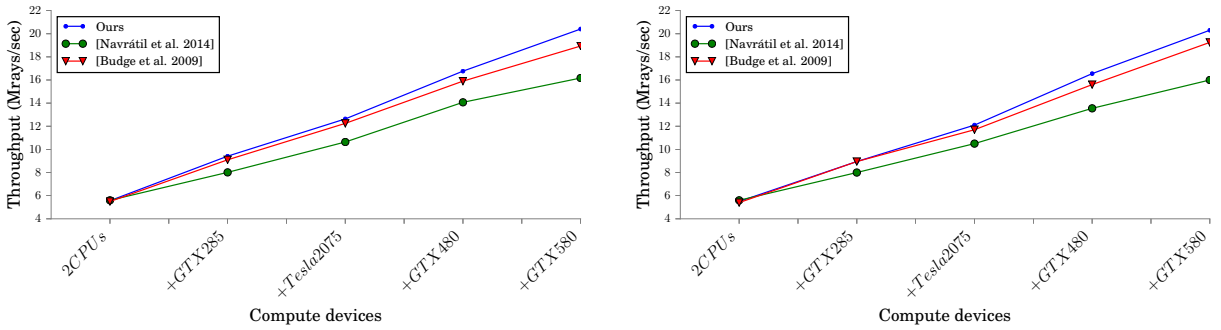
In Table 2, we have measured the rate of such prediction misses, and the average peak throughput improvement compared to a non-prediction case where job prediction is disabled. *Boeing777* causes higher rates of predicting mis-fetching jobs due to the non-uniform distribution of rays and high model complexity. Nonetheless, it is clear that our prediction miss rate is reasonably low (e.g., less than 40 %) and the benefit of our job prediction exceeds its overhead caused by incorrect predictions.

Per-bounce analysis. It has been well known that as rays get more incoherent, it becomes harder to maintain a higher throughput and bigger data granularity for each ray queue [Pharr et al. 1997]. In the *SponzaMuseum* scene where indirect illumination dominates, it tends to generate many incoherent rays and thus a lot of small ray queues scattered across the scene, which in turn increases the I/O overhead and reduces the granularity of each ray queue.

To look deeper into the issue of incoherency of rays, we measure the throughput as a function of path lengths in the *SponzaMuseum* scene. One can easily expect that the level of incoherency goes higher as we have longer path lengths. As Fig. 4 shows, our method shows lower throughputs as we have longer paths. Nonetheless, our method shows graceful performance degradation, especially compared with the prior work. This graceful degradation of our method is mainly attributed by better prefetching more *SceneData* as more **(Shadow)RayIntersect** jobs are queued, increasing the effect of latency hiding. Compared to ours, the prior work cannot increase the fetching rate as the ray queues get scattered.

HPG '17, Los Angeles, CA, USA

978-1-4503-5101-0/17/07…$15.00
DOI: 10.1145/3105762.3105784



(a) Boeing777 (b) SponzaMuseum

Figure 1: This figure shows the throughput graphs of each method as more computational resources are added to the system consisting of a single multi-GPU workstation. The X-axis represents the composition of compute devices in the workstation, where Y-axis represents throughput in the total number of rays shot per second.

Type	CPU	Main memory	GPU Memory	GPU	Note
A	i7-4770K 3.5GHz octa-core	DDR3 8GB	6GB	GTX Titan	1GbE LAN, 4 nodes
B	i7-4790K 4GHz octa-core	DDR3 8GB	6GB	GTX Titan	
C	Xeon E5-2690 2.9GHz 16-core	DDR3 8GB	6GB	GTX Titan	
D	Xeon E5-2690 2.6GHz 16-core	DDR3 8GB	6GB	GTX Titan X	
R	i7-3770k 3.5GHz quad-core	DDR3 8GB	4GB	GTX980	

Table 1: Type of nodes in our experimental setup

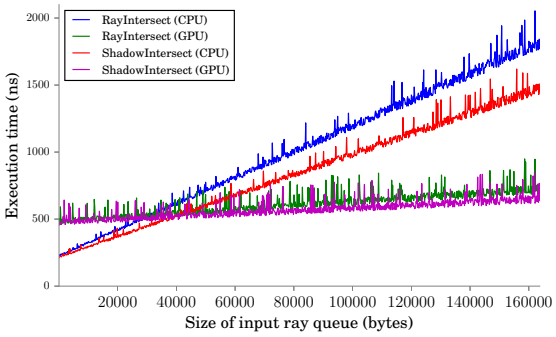


Figure 2: This figure illustrates the measured execution time for ray tracing jobs w.r.t. the size of ray queues. It shows the linear behavior that fits well into our timing model. Timing models for other inputs follow the similar trend.

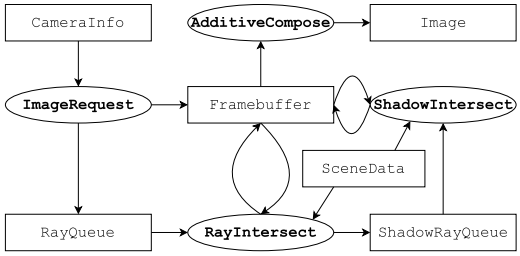


Figure 3: An illustration of dependencies for the path tracer. The circular shapes with the bold font are jobs, while the rectangular ones are associated data.

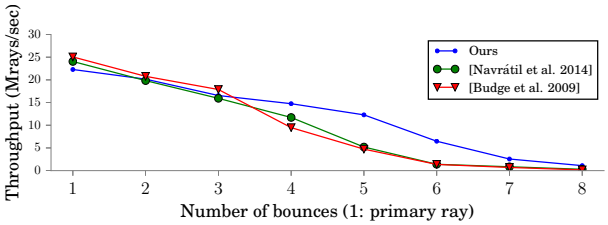


Figure 4: This figure shows the throughput breakdowns of different methods as a function of the number of bounces of rays in *SponzaMuseum* scene in heterogeneous 8-node setup.

Scalability analysis over a workstation. In this scenario, we have one workstation to serve the rendering requests. The workstation consists of two Intel Xeon 2.93GHz CPUs and 8GB RAM and four different GPUs (GTX285, Tesla 2075, GTX480, GTX580). All GPUs use 2GB of their internal memory as buffer. To show the increasing horizontal scalability, we begin with using only two CPUs, and adding GPUs one by one. The result is shown in Fig. 1.

Compared to 8-node cluster setup, the communication cost is much lower since LAN channel is not used in this setup. In turn,

the data fetch optimization has relatively less effect. Nevertheless, our scheduler still outperforms prior methods. Unlike cluster setup where the fetch path optimization drives the improvement, we have observed that data pre-fetching is the key component to the performance in this setup. Compared to the prior methods, our system hides fetch latency by 27% for *Boeing777* and 23% for *SponzaMuseum* in average when all GPUs are utilized, which in turn results in up to 8 % total throughput improvement over [Budge et al. 2009], which is also designed to be optimized for the workstation setup.

Function Schedule(*DCG, J*)
  Data: DCG: the device connectivity graph
    J: the list of jobs
  Result: A map of schedule p_d for each $d \in D_{target}$ where $D_{\text{target}} := D_{\text{compute}} \cup E_{\text{DCG}}$
  // Initialization
  foreach *d in* D_{target} **do**
    Initialize p_d with an empty schedule
  end foreach
  // Iterative job allocation
  while $\min_{d \in D_{compute}} EndTime(p_d) < k \wedge J \neq \emptyset$ **do**
    $d := \arg\min_{d \in D_{\text{compute}}} EndTime(p_d)$
    j, $s_{fetch} := NextJob(\text{DCG}, d, p, J)$
    Merge dependent T-jobs s_{fetch} to p
    // Schedule process on p_d
    $t_{max} = EndTime(s_{fetch})$
    Add the job j at $p_d[t_{max}, t_{max} + T_{EXEC}(d, j, W(j))]$
    Remove j from J
  end while
  return *all the schedules p*

Algorithm 1: GMB Algorithm - Scheduling module

Function NextJob(*DCG, d, p, J*)
  Data: DCG: the device connectivity graph
    d: the compute device to schedule a job
    p: the current schedule
    J: the jobs to select
  Result: j: the next job to execute
    $datapath$: the list of paths in DCG
  $j_{\text{next}} := \emptyset$
  $t_{\emptyset} := \infty$
  foreach j *in* J **do**
    $W_j := RequiredDataBlocks(j)$
    foreach w *in* W_j **do**
      $path_w, t_w := FindPath(\text{DCG}, Mem(d), p, w)$
    end foreach
    $t_j := max_{w \in W} t_w$
    if $t_{j_{next}} > t_j$ *or* $(t_{j_{next}} = t_j$ *and* $\frac{T_{EXEC}(d, j_{next}, W_{j_{next}})}{T_{SETUP}(d, j_{next})} < \frac{T_{EXEC}(d, j, W_j)}{T_{SETUP}(d, j)})$ **then**
      $j_{\text{next}} = j$
    end if
  end foreach
  return j_{next}, $ConstructFetchSchedule(path_{W_j})$

Algorithm 2: GMB Algorithm - Job selection

REFERENCES

Brian Budge, Tony Bernardin, Jeff A. Stuart, Shubhabrata Sengupta, Kenneth I. Joy, and John D. Owens. 2009. Out-of-core data management for path tracing on hybrid resources. *Computer Graphics Forum (EG)* 28, 2 (2009), 385–396.

Mark Harris. 2012. How to Overlap Data Transfers in CUDA C/C++, https://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/.

Adam Osborne. 1980. An Introduction to Microcomputers: Volume 1âĂŤBasic Concepts, Berkeley: Osborne. (1980).

M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan. 1997. Rendering complex scenes with memory-coherent ray tracing. In *ACM SIGGRAPH*. 101–108.

Function `FindPath`(DCG, m_d, p, w)
  Data:
    DCG: the device connectivity graph
    m_d: the target memory device to transfer a data block
    p: the current schedule
    w: the data block to transfer
  Result: $path$: the fetch path for w from sources to m_d
    $time$: the time to finish fetch
  `// Initialization`
  Initialize Q with empty min-priority queue
  Add vertex S to DCG
  foreach d *in* DCG **do**
    if $d = S$ **then**
      $t_d := 0$
    else
      Add edge $d \rightarrow S$ to DCG
      $t_d :=$ inf
    end if
    $prev_d := \emptyset$
    $Q.push(d, t_d)$
  end foreach
  `// Shortest path search loop`
  while Q *is not empty* **do**
    $u = Q.popMin()$
    if $u = m_d$ **then**
      return $[S \rightarrow \ldots \rightarrow prev_{m_d} \rightarrow m_d], t_{m_d}$
    end if
    foreach v *in* $adjacent(u)$ **do**
      if $u = S$ **then**

$$new_t := \begin{cases} 0, & \text{if } w \text{ is already loaded at } v \\ t, & \text{if } Load(w) \text{ is found at } t \text{ in } p_v \\ \infty, & otherwise. \end{cases}$$

      else
        $new_t := max(t_u, EndTime(p_{u \rightarrow v})) + T_{TRANS}(u \rightarrow v, w)$
      end if
      if $new_t < t_v$ **then**
        $t_v := new_t$
        $prev_v := u$
        $Q.setPriority(v, t_v)$
      end if
    end foreach
  end while

Algorithm 3: GMB Algorithm - Fetch path construction