High Performance Graphics (2010)
M. Doggett, S. Laine, and W. Hunt (Editors)

Task Management for Irregular-Parallel Workloads on the GPU

Stanley Tzeng[†], Anjul Patney and John D. Owens

University of California, Davis

Abstract

We explore software mechanisms for managing irregular tasks on graphics processing units (GPUs). We demonstrate that dynamic scheduling and efficient memory management are critical problems in achieving high efficiency on irregular workloads. We experiment with several task-management techniques, ranging from the use of a single monolithic task queue to distributed queuing with task stealing and donation. On irregular workloads, we show that both centralized and distributed queues have more than 100 times as much idle times as our task-stealing and -donation queues. Our preferred choice is task-donation because of comparable performance to task-stealing while using less memory overhead. To help in this analysis, we use an artificial task-management system that monitors performance and memory usage to quantify the impact of these different techniques. We validate our results by implementing a Reyes renderer with its irregular split-and-dice workload that is able to achieve real-time framerates on a single GPU.

1. Introduction

Current hardware-accelerated graphics pipelines exploit ample parallelism, both regular and irregular. Historically, the programmable parts of the pipeline—vertex and fragment shaders—have been fairly regular, generally following a single-primitive-in, single-primitive-out model. Newer programmable stages, such as geometry shaders, can target more irregular parallelism. Yet the complexity of *managing* this irregular parallelism is invisible: the details of work distribution across parallel units, scheduling, work queuing, load balancing, and memory management are all hidden from the programmer. Just like current irregularly-parallel fixed-function units (such as rasterizers and clippers), the GPU's hardware, assisted by low-level drivers, handles these details with special-purpose, customized hardware queues and schedulers.

Today, the increasing general-purpose programmability of the GPU is allowing developers to explore *programmable pipelines* [Pha06] in which programmers specify not only individual stages but also the overall structure of the pipeline. Both industry and academia are racing toward providing the hardware, tools, and software prototypes to make such custom pipelines possible and usable [SCS*08, LHLW10, ZHR*09]. Crucially, implementing these programmable pipelines moves the complexity of managing parallel workloads from the *hardware* onto *software*, from vendors to programmers. Our work focuses on how to support these parallel workloads, particularly full pipelines featuring irregularly-parallel workloads, on today's programmable GPUs.

Informally, we see a future where developers can use a wide selection of stages in constructing programmable pipelines: texture samplers, tessellators, compositors, and so on. These developers can use existing stages or write their own. If we think of these stages as bricks, we are interested in providing the mortar between these bricks: how can we assemble a set of parallel tasks into a pipeline that efficiently utilizes the resources of the hardware in providing a high-performance implementation? Such a contribution would not be limited only to programmable graphics pipelines but also target irregularly parallel computing problems like adaptive multiresolution data structures or tree and graph traversal and construction.

The challenges of such a task management system include

† {stzeng, apatney, jowens}@ucdavis.edu

Figure 1: *This diagram shows images rendered at 800×800 using our CUDA-based Reyes renderer. The Killeroo (left) and bubbles (right) were rendered with 16 jittered samples per pixel, and the teapot (center) was rendered with 42 jittered samples per pixel. Killeroo and bubbles render at approximately 20 frames/sec, while teapot with motion blur achieves 3 frames/sec.*

the following: (1) work distribution to parallel units; (2) load balancing across these parallel units; (3) work scheduling; and (4) memory management, particularly in the context of the modest amount of memory on modern GPUs. These goals are particularly challenging because we target our system not to conventional processor architectures but instead to the data-parallel GPU, which lacks comprehensive hardware support for native task parallelism. Our solution to these challenges is a parallel task queuing system combined with a pipeline authored in an uberkernel/persistent-kernel style. Our system provides the basis of a robust infrastructure for inter-stage data queuing by efficiently managing irregular tasks in programmable pipelines.

We evaluate our system on both a synthetic work generator, capable of modeling various kinds of irregular workloads, and on the Reyes graphics pipeline [CCC87], a high-quality rendering pipeline originally designed for cinematic rendering. Our contributions are as follows:

- In the context of our parallel task queue implementation, we analyze different load-balancing schemes for the GPU and their effect on different types of workloads. We show that a task-donating scheme is the most suitable for workloads with irregular parallelism.
- We demonstrate real-time performance on simple Reyes workloads by using our dynamic task management scheme. In contrast to previous work, our implementation schedules all irregular workloads on the GPU.

2. Previous Work

Parallel Work Management Dynamic work-management in parallel systems is often achieved through queues that contain information about pending work items. In the simplest setup, a single queue contains all work items. Execution units access this *block queue* in parallel to either fetch work items for execution, or append new work items based on past execution. In 2009, Aila and Laine [AL09] discussed such a setup in the context of ray tracing. A monolithic queue, however, tends to introduce a single point of synchronization for all parallel units, resulting in inefficient queue transactions as the number of parallel units grows. In general, previous research that manages irregularly parallel work [AL09, PEO09, ZHR*09] uses monolithic queues and would benefit from the techniques that we propose in this paper.

Load Balancing, Task Stealing, and Donation One way to avoid the synchronization inefficiencies of a single queue is to divide it into multiple parts, one for each execution unit. Now each unit has exclusive access to its queue, without any requirement for a single synchronization point. However, irregular parallelism of work items leads to load imbalance, as some queues will empty faster than others, leaving many processors idle while work is still remaining.

As a result, some sort of load-balancing is critical in a distributed queuing scenario. In our work, we draw many inspirations and insights from early work done on CPU multiprocessor and multithreaded programming. Work stealing is a popular multiprocessor scheduling scheme and we refer interested readers to the seminal work done by Blumfe and Leiserson [BL99]. Work sharing [HS08] or work donation is another scheduling scheme which is made popular by OpenMP [CJvdP07]. While work stealing is important in ensuring that no processors go unused, work sharing can not only help distribute load even while all processors are busy but also reduce overall memory usage. These core load balancing schemes are crucial in understanding how newer programmable graphics pipelines can maximize throughput.

In the context of graphics, parallel ray tracing was some of the first work to require load balancing. Heirich and Arvo explore parallel ray tracing across CPU clusters and examine how different load balancing schemes designed for clusters affects ray tracing efficiency [HA98]. More recently, work management in NVIDIA's interactive ray tracer OptiX [PBD*10] provides an example of a GPU-based application with dynamic load balancing. Load balancing on the GPU has become a recent topic of interest with the advent of more sophisticated synchronization operations such

as atomics. Cederman and Tsigas implemented four load-balancing techniques in the context of a single irregularly-parallel task, octree partitioning [CT08], concluding that their implementation of Arora et al.'s CPU-based task-stealing method [ABP98] had the best performance and scalability. Their excellent work was the first to introduce more sophisticated queue management techniques to the GPU, though their design decisions are substantially different than ours (Section 3). The GRAMPS programming [SFB*09] model for graphics hardware is an abstraction on how to design different graphical pipelines on data parallel systems. Aila and Laine [AL09] explored the efficiency of block-queue-based work distribution in a GPU-based ray tracer, concluding that a persistent-thread approach rather than hardware work distribution was the most efficient.

Reyes on the GPU Programmable GPUs are becoming increasingly suitable platforms for implementing non-standard rendering strategies, and have been employed for research in real-time Reyes rendering. Most notable is Zhou et al.'s software Reyes renderer, RenderAnts [ZHR*09], which maps all stages of Reyes to a GPU. RenderAnts is an order of magnitude faster than RenderMan, but does not run at interactive rates. It features a novel CPU-based task scheduler for dicing, shading and sampling, a shader compiler with out-of-core texturing, and multi-GPU rendering capabilities. RenderAnts, however, does not address task management on the GPU, especially for the irregular parts of the pipeline like surface subdivision.

Other researchers have concentrated on GPU implementations of specific stages of the Reyes pipeline. Patney et al. [PO08, PEO09] explored smooth surface subdivision for a real-time GPU-based Reyes renderer, using a centralized queue for work distribution. Fatahalian et al. [FLB*09] and Eisenacher and Loop [EL10] studied rasterization of micropolygons on a massively-parallel platform. Fisher et al.'s DiagSplit [FFB*09] extends Reyes-style surface subdivision to a crack-free scheme, and exploits tessellation hardware in the Direct3D 11 pipeline to achieve high performance. Despite the volume of literature in this area, there has been little focus on work management for irregular Reyes stages. In this paper we attempt to address this problem as it is both interesting and crucial to understand how to map irregular workloads , such as Reyes, onto a GPU programming model.

3. Dynamic Task-Management on a Modern GPU

A software task scheduler must fulfill the same goals as its hardware counterpart: it should optimally utilize processor resources while minimizing memory usage. We wish to design a task management system that can keep data flowing through pipeline stages while maintaining maximum processor utilization and balance. In this section, we discuss our design decisions of how we implement GPU dynamic task management in the context of work granularity, processor utilization, scheduler emulation, and memory management. Returning to the brick and mortar analogy, this section discusses the ingredients of our mortar recipe and our justifications for our design choices.

At a high level, our system is structured as a work queue. Work units, which can be either regular or irregular, and which can come from a single task or many tasks, are placed in the queue. Processors then consume these work units, possibly producing additional work units that are added to the queue. We make four key design decisions in our implementation that together distinguish our work from previous systems: (1) work units of the size of a warp; (2) the use of persistent kernels to better address irregularly parallel work; (3) uberkernels to eliminate the overhead of switching kernels; and (4) a task-stealing and -donating queuing system to minimize memory usage.

Warp size work granularity While GPU programming models typically use a single thread/work-item as the core unit of parallelism, we instead choose the size of a SIMD-group (known in the CUDA programming model as a *warp*). Our task queues store work in warps, not individual threads, and we dispatch work in warp-sized chunks. Blocks greater than one warp require explicit synchronization to ensure proper shared memory accesses and this can be hard to achieve within control flow statements. Since warps run in lockstep, this obivates any requirement for explicit thread synchronization and can help with more efficient coding. Secondly, we can now view a warp as a single MIMD thread. This granularity allows us to maintain the efficiency of SIMD execution (within a warp) while granting us MIMD-style granularity of warp-sized independent operations. In other words, we can view the hardware that executes a SIMD-group as a *processor* and throughout the rest of this paper we will use this term.

Persistent thread scheduler emulation Modern GPU programming environments have poor support for kernels that generate and then consume irregular amounts of work. The static model of GPU execution would require programmers to run one kernel for each pass through the work, requiring a global barrier between each kernel and likely a round-trip to the CPU to size and call the next kernel iteration.

An alternate approach, and the one we adopt, is the *persistent threads* model [AL09]. This model launches only enough warps to fill the machine but leaves those warps alive throughout the entire kernel; each warp thus processes many work units. Persistent threads can append work to the end of a queue and then consume work from the front of the queue, all within the same kernel call; thus avoiding the overhead of a global synchronization. Persistent threads are well-suited for efficient production and consumption of irregularly-parallel work and we leverage them in our implementation.

Uberkernels for processor utilization The ideal scheduler keeps each processor as busy as possible throughout the execution of the pipeline. Traditional methods of mapping one pipeline stage to one GPU kernel introduce explicit barriers between pipeline stages, which hinders processor utilization. Furthermore, since our focus is on pipelines with irregular workloads, we cannot predict where this barrier will occur. We wish to eliminate this barrier altogether and allow work that has been processed in one stage to flow directly to the next.

Our solution is to combine multiple pipeline stages into one kernel with the uberkernel programming model [TK09]. Uberkernels pack multiple different tasks into a single physical kernel, expressing task parallelism within that one kernel without the overhead of switching between kernels. In an uberkernel, a work unit can be processed by one path of the uberkernel, pushed back onto the input queue, and then processed down another path of the uberkernel, effectively going through multiple stages of the pipeline within one kernel.

Task donation memory management GPUs do not have the same amount of memory as CPUs, so a task management system must rely only on a modest amount of memory, especially if the queues are required to be stored in on-chip memory. However, since we are dealing with irregular workloads, we must handle the case when the memory allocated to a specific processor runs out and must spill over to the next. Our memory management system is a distributed queueing scheme in which each processor has its own private dequeue (a *bin*). When a processor's bin is empty, the processor may take work out of another processor's bin (*task stealing*) to avoid idling that processor. In addition, when its own bin is about to overflow, the processor may push work out of its own bin into another processor's bin (*task donation*). The result is less memory per bin than a system that lacks donation.

Summary We combine our four design decisions to build a task-stealing and -donating scheduler that can support multiple pipeline stages within a single kernel. Our scheduler runs with a granularity of 32 threads, corresponding to the SIMD width of our target machine, and can support as many pipeline stages as resources allow. Our task-donating scheme reduces the overall memory usage for our queues.

The combination of persistent threads and uberkernels directly addresses the problem of high processor utilization on irregular workloads. Persistent threads allow us to cycle work multiple times through a kernel by consuming from and writing to the same queue; they are thus well-suited for irregular workloads in pipelines. Uberkernels allow task divergence in execution routine behavior, allowing processors to switch between executing two different stages of the pipeline at once and also eliminating the need for a global barrier between pipeline stages.

To enforce accuracy and performance, we enforce two extra constraints on each block:

Coherence All threads in a warp must execute the same path. Threads executing one branch of the uberkernel cannot be in the same warp as threads executing a different branch.

Contingency Only one block of threads has access to a unit of work at a time. A work unit cannot be processed by two processors at once.

We achieve our coherence constraint by scaling the size of each block such that one block contains exactly one warp of threads. The contingency constraint is maintained by the use of atomic operations that synchronize between blocks fetching and writing data.

For our testing purposes in the synthetic work scheduler (Section 4) and the Reyes pipeline (Section 4.4), we used two distinct kernels (split and dice) within the same uberkernel, which is well within the resource limits on a modern graphics card.

4. Implementation

In this section we discuss the load balancing schemes that we investigate in this paper followed by implementation of our Reyes renderer. We encapsulate each scheme in our synthetic work generator so that we can measure performance and memory footprints under different workloads.

We are given n processors $p_1 \dots p_n$ and initially q work units $w_1 \dots w_q$ stored in global memory as input. The output is an output queue o that stores work which has been processed. Each work unit may generate more work with characteristics determined by the synthetic work generator.

4.1. Synthetic Work Generator

Our synthetic work generator takes in, for each processor, a random number $x \in [0,1]$, an exit probability p, and the spawn amount m. The spawn amount is the amount of work that this work unit will spawn. Upon fetching a unit of work, the processor tests if $x > p$. If so, then the current unit of work is flushed out of the system into the output queue o. If not so, then the processor will push $m-1$ units of work back onto the input queue while holding onto one work unit itself so it does not need to fetch from the queue again. We define a *trial* as the process of all n processors processing the q units of work and the work that any work unit may spawn. An *iteration* of the trial is defined as one processor processing one unit of work. At the end of the trial the input queue is empty and all work units are flushed out to o.

4.2. Queuing Strategies

Block Queuing The block queuing model consists of a global memory dequeue d where processes remove work

from d through the tail and insert work through the head. One single lock l_{head} locks the head of d for exclusive read and write access. Processors are free to read from the tail of the dequeue up until when d contains exactly one work unit. In this situation, every processor that wishes to read the final work unit must grab l_{head}. The processor that successfully obtains l_{head} reads the final unit of work from d and every other process that queries d will see that there is no work left.

While block queuing is the simplest and most straightforward to implement, it has heavy lock contention. This occurs when many processors want to write back work into d at the same time. Adding data back into a block queue is a serial process since there is only one lock.

Distributed Queuing In the distributed queuing scheme, instead of a global dequeue, each processor has its own private dequeue (a *bin*) $d_1 \dots d_n$ with a lock $l_1 \dots l_n$. We distribute the initial work evenly into each bin. Each processor has complete access to its own dequeue and thus no locks are needed. All bin sizes are equal. The problem with distributed queuing is that it is static. Once a processor has processed all the work in its bin, it idles. The distributed queuing scheme is held back by the slowest processor.

Task Stealing Task stealing is an extension to the distributed queuing scheme which allows communication between processors via locks to increase processor utilization. When processor p_i's bin is empty, p_i attempts to retrieve work from another processor's bin. This is *stealing* and p_i can only steal from the head of another bin. Each processor comes with its own lock l_i which is used to lock the tail end of its bin.

In the task stealing scheme, a processor has exclusive access to its head for read and writes; the head is used for stealing. When a workload spawns more work, the processor pushes the new work onto its own tail, and it only pushes the work onto the head when it detects that there is no more room in the tail. One can think of the dequeue's tail being a stack used only by its processor. When a processor attempts to steal work it may only do so from the victim dequeue's head, in which case it must acquire the victim processor's lock first.

Lock contention is significantly less than the block queuing case. Locks are only used in two conditions: when p_i attempts to steal work, or when the extra work generated by p_i cannot fit into the tail, in which case p_i writes the work back into the front. The head of d_i is either read from or written to respectively. Victims are chosen in a round robin fashion; while there are improvements to the bound done by randomness [BL99], a round robin fashion was simple enough for our purposes.

The drawback to task stealing, as with distributed queuing, is the memory usage—task stealing cannot handle bin overflows. Each bin must have enough memory to fit the worst case processor workload. If bin d_i cannot hold the work generated by p_i, then our solution is to stall p_i until another processor p_j steals work out of d_i. However, this can only occur if d_j is empty and on a well-balanced system, this condition may not occur until many iterations into execution. Our variation of this work sharing scheme, which we refer to as *task donation*, is a solution to this problem.

4.3. Task Sharing / Donation

The task donation scheme is designed using our design choices from Section 3. In the regular work sharing scheme, p_i will randomly choose another processor p_j and their respective bins d_i and d_j will have their work balanced out (hence the term *sharing*). This scheme allows full bins to offload work to other bins. However, such a scheme does not adapt well onto the GPU due to lack of locality of processors (i.e. accessing any bin is a trip to global memory). Instead we adapt the following scheme: when processor p_i detects that it has spawned extra work which cannot fit into d_i, then it selects a receiver processor p_r and then p_i will "donate" its extra work to p_r. To do so, p_i acquires the lock l_r and write the work to the head of d_r. Like task stealing, p_i chooses p_r in a round robin fashion.

Task donation is a solution to bin overflowing, for it allows full bins to spill work to other bins. This means that this scheme can use a much lower bin size than task stealing, which is especially important if work queues need to be maintained in on-chip memory. However, this solution comes at the cost of more lock usage. Each time a processor donates, it may only do so after acquiring the lock to the receiving processor. Although there is a higher contention for locks, our results show that the task donation scheme performs better than task stealing when finding the balance of memory vs. performance (see section 5).

4.4. Reyes Pipeline

In this subsection, we discuss how principles of dynamic load balancing apply to a Reyes rendering pipeline. Figure 3 shows a basic diagram of a GPU-based Reyes renderer. As mentioned previously, Reyes is a high quality rendering pipeline extensively used in cinematic applications [CCC87].

4.5. Design of Individual Pipeline Stages

We implemented individual stages of the Reyes pipeline around the previously described queuing infrastructure, taking into account the available queues for individual stages. Except for subdivision which is a combination of split and dice, each stage is implemented as a separate CUDA kernel that occupies the entire GPU at once, and operates over elements either taken from a queue or as forwarded from the

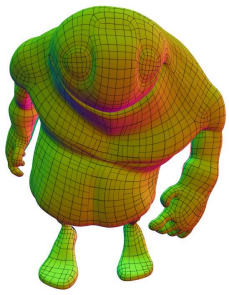


Figure 2: *This figure shows the surface patches generated after Reyes subdivision. During this operation, smooth surface patches are adaptively subdivided to smaller ones, which are diced to pixel-sized micropolygons. This recursive operation results a very irregular execution pattern where every unit is capable of producing different amounts of work.*

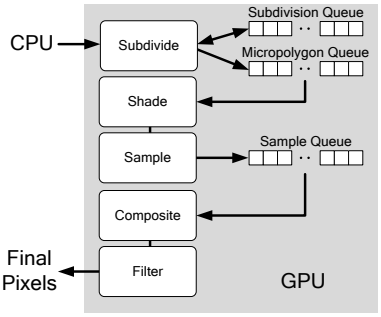


Figure 3: *Basic diagram of the prototype Reyes renderer. Note that the system is based on three main queues for subdivision, output micropolygons, and sampling. Of these, the subdivision queue is read-write, while others are write-only.*

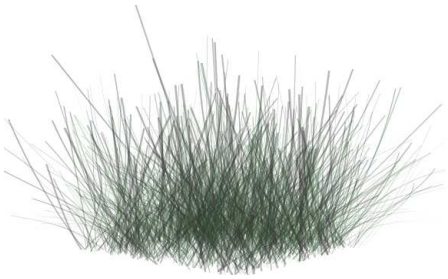


Figure 4: *1600 blades of grass (each with $\alpha = 0.4$) rendered using our renderer. At a resolution of 800×800 pixels and 16 samples per pixel, this scene runs at approximately 20 frames per second on a single NVIDIA GPU.*

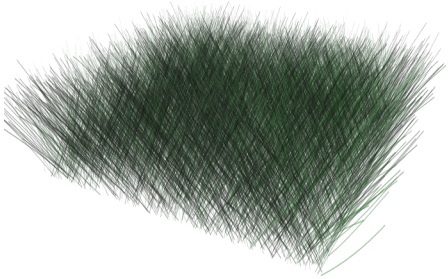


Figure 5: *10,000 blades of grass ($\alpha = 0.4$) rendered using our renderer at a resolution of 800×800 pixels and 16 samples per pixel. This scene runs at approximately 0.7 frames per second on a single NVIDIA GPU.*

previous stage. The justification for this is twofold: firstly, stages of shading and sampling are individually expected to be sufficient to fill the device, and secondly, they are much more regular than subdivision, since they are not usually expected to involve dynamic work creation. Also, there is a hard synchronization requirement after sampling; i.e. they must operate in strict sequence.

Block queuing is sufficient for the more regular stages of the pipeline, that are either one-to-one mapped (shading), irregular read-only (composite/filter) or irregular write-only (sampling). However, subdivision involves irregular reading as well as writing. Hence, as the most irregular stage in this pipeline, we focus in depth on Bound/Split and Dice.

We perform *subdivision* in an GPU kernel that performs either split or dice on the work items obtained from the dynamic subdivision queue. In this way, it is able to both create work (split primitives) and output result (diced micropolygons) as a part of the same kernel. Execution is organized into groups of 32-wide SIMD-coherent threads, which operate on individual bins of work. Whenever a group runs out of work in its bin, it steals work from another non-empty bin. Split and dice are parallelized over the 32 SIMD lanes in order to maximize efficiency of the operation.

We perform micropolygon *shading* in a single pass over all grids. Although shading in an actual application is expected to be complex and dominate the frame time, in this paper we consider Phong and other simple shaders. Shading is efficiently parallelized across the SIMD width of a GPU.

In our rendering system, we implement *sampling* as a parallel kernel over all available micropolygons. In parallel, the sampler evaluates bounds for given micropolygons, and performs intersection tests with the samples contained. The sampler uses an atomic counter to update a write-only sample queue for output samples.

We implement *composite and filter* over the collected samples through a sequence of CUDA kernels that evaluate final colors for screen pixels. We first sort the generated samples to corresponding subpixels in increasing order of depth, and then for each subpixel in parallel, we blend the respective samples. This is followed by a simple filtering kernel

that uses blended sample values to calculate final pixel colors and writes them to the output pixel buffer.

5. Results

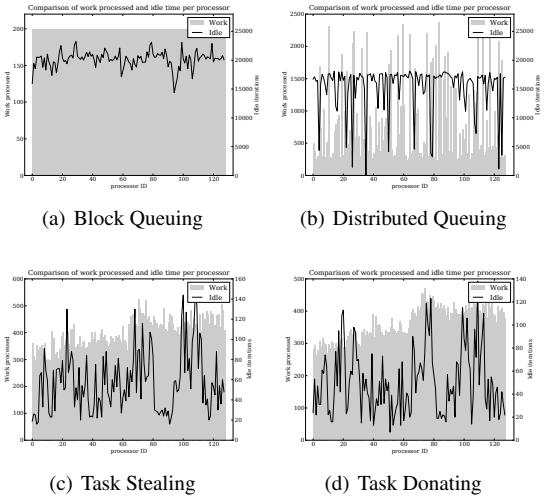
Experimental Setup We measure the performance of our scheduling routines by looking at several factors: utilization in terms of processor idle time, load balance in terms of work units processed per processor, and memory usage in terms of memory used versus donation. We will discuss why these factors are relevant to our setup.

Processor idle time measures two things: how long a processor waits for a lock, and the number of iterations that the processor is idle. The latter is a measure of processor utilization, for the less a processor is idle, the more work it is processing. We measure performance (throughput) by examining the total amount of work units processed for one run over the time it took to process it. The resulting number is a good indicator of the overall efficiency of a task-management scheme. To effectively measure the memory usage benefit of task donation, we examine its performance over varying bin sizes. The smaller the bin size, the more likely a processor will overfill its bin and have to donate; this implies a slower overall runtime.

Synthetic Work Generator Figure 6 shows the analysis of our load balancing in terms of work processed with idle time drawn over. From the results, we can see that when there is a single block queue, all processors process the same amount of work. Block queuing has perfect load balancing, but at the cost of an alarming amount of wasted time, due to high contention for the lock to write data back to the head. Distributed queuing also has high idle time for a different reason: it has no contention but high wasted time due load imbalance. Most processors wait for only a few processors to finish. Note only a single processor has no idle time. Task stealing and donation show similar performance characteristics, but with more than two orders of magnitude less idle time. Stealing and donation are excellent methods to maintain high processor utilization.

Figure 7 shows the analysis of performance as we use p to vary the workload characteristics (see section 4.1) from highly regular to highly irregular. We observe that as the workload becomes more and more irregular, performance of a single block queue is significantly slower than that of task stealing or task donation. As workload is more irregular more work units must be written back to the queue and this creates high lock contention.

As mentioned before, task donation permits much lower bin sizes, which is extremely important for implementations where queues need to remain on on-chip memory, perhaps in a cache. But what is the cost of ensuring on-chip access? Figure 8 shows the impact of task donation on the performance and memory behavior for an irregular rendering workload.



(a) Block Queuing (b) Distributed Queuing

(c) Task Stealing (d) Task Donating

Figure 6: *Synthetic work processed as a bar graph in grey with processor idle time as a line graph in black. Note that the two graphs on top have a different idle time scale (a factor of 150 times greater) than the graphs on the bottom. This is because there is much more idle time in the block queue due to lock contention and there is much more idle time in distributed queuing due to all processors waiting for one processor to finish. With task stealing and task donation, there is much less idle time due to processors being to steal work to keep themselves busy.*

We simulate a scenario with increasingly constrained upper-limit of bin sizes, and redistribute overflowing bins using either the proposed Task Donation or an approach based on dynamic queue resizing, which we achieve using an additional overflow buffer. Rather than resizing the queues on the GPU, which at the time of this writing is not supported, the overflow buffer servers as a block queue where processors with their bin full can offload work into and other processors can steal from there.

As bin sizes are reduced, we see that dynamic queue reallocation suffers from significant performance overheads, due to both imbalance in bin sizes and extra synchronization required in the overflow buffer. Task Donation does not require such extra synchronization and still manages to maintain balance in bin sizes. As a result, subdivision time using task donation is only slightly affected.

Reyes Our GPU implementation of work management is able to achieve real-time performance for most scenes tested. Table 1 shows statistics for some of the scenes tested. Figures 1, 4 and 5 show the obtained images. We are able to achieve interactive framerates for scenes with up to 16–30 jittered samples per pixel. Detailed performance of the subdivision step, which has been most extensively studied, com-

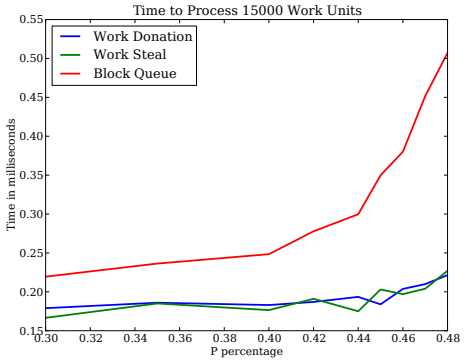


Figure 7: *Comparison of different task management schemes' performance against changing p with a fixed $m = 2$. p varies from a low value (high work regularity) to a high value (high work irregularity). We keep $p < 0.5$ to ensure that the system is stable and the trial will end. We measure the time it takes for each load balancing scheme by clocking how long it takes to process 15,000 work units. Task stealing and task donation give relatively equal performance, but as the work becomes more irregular, block queueing takes longer. This is due to the higher lock contention that comes along with the block queueing scheme.*

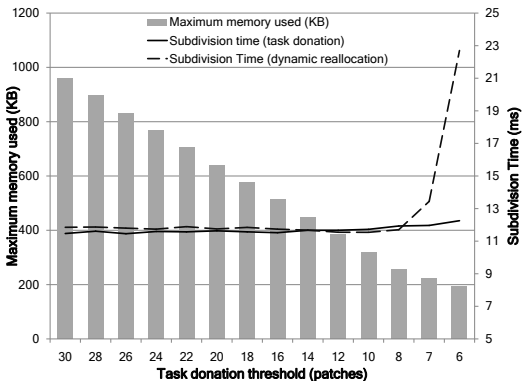


Figure 8: *This plot quantifies the overhead of limiting bins to fixed sizes. For an irregular rendering workload (teapot subdivision), we show the impact of task donation on the memory and performance behavior, and contrast it to an approach based on dynamic reallocation of queues that overflow. Going from left to right, we decrease the threshold bin size for task management. We notice that as the bin sizes are increasingly constrained (perhaps by the size of on-chip memory), time taken for subdivision based on donation is only slightly affected. However, with dynamic reallocation, subdivision time increases significantly for small bin sizes.*

SCENE	PRIMS	GRIDPTS	SAMPLES	PERF (frames/s)
Grass	5.2K	446K	875K	19.30
Killeroo	11.5K	330K	649K	18.48
Bubbles	660	479K	1.46M	16.75
Teapot	268	218K	689K	23.35
Teapot (motion)	268	218K	690K	3.11
Big Guy	3592	450K	1.36M	11.05
Big Grass	74K	9.5M	7.97M	0.70

PRIMS	Total number of rendering primitives (after subdivision)
GRIDPTS	Total number of shading points (after subdivision)
SAMPLES	The number of subpixel samples (before composite)

Table 1: *Rendering performance for the various scenes tested. Each scene was rendered at 800×800 pixels with 16 samples per pixel. Note that for the teapot, introducing motion blur causes negligible change in the number of samples, but the performance is significantly low. This is due to the rise in the number of samples tested under motion.*

pares favorably to previous work: we generate 100–256M micropolygons per second (up/s). In comparison, Fisher et al. [FFB*09] achieve 129.9M up/s on an 8-core Intel CPU.

6. Discussion

From the results in section 5, we are able to observe the advantages of our task management strategy for irregular workloads. Firstly, from Figure 6 can see that distributed queuing is better than block queuing due to a significant reduction in the contention for memory locks. Secondly, a simple distributed queue is not sufficient, because it wastes execution cycles due to load imbalance, as most processors end up waiting for a few to finish their work. Memory footprint is also high, especially when on-chip memory is limited. Task stealing significantly reduces the idle cycles for waiting processors by fetching work from busy ones. Task donation further improves the situation by essentially preemptively stealing a task from a full queue, thereby reducing the maximum memory footprint of the distributed queues. The resulting system is a robust work manager that can efficiently schedule workloads with a high degree of irregularity.

7. Conclusion

The above results are based on the assumption, characteristic of currently available hardware, that atomic operations and mutual exclusion locks are expensive constructs. Though these operations are inherently serial, newer hardware may mitigate this cost; in particular, the latest AMD GPUs are known to have high-performing atomic operations. Also, GPUs like NVIDIA Fermi and Intel Larrabee have on-chip

caching, which allows for fast resolution of atomic operations, making them and consequently locks much cheaper. This change in our assumption, depending on its magnitude, could make block queuing a much better alternative than its current form. However, it also makes task stealing and donation cheaper, and block queuing still suffers from contention for a single lock, which is likely to continue to affect performance as chip-level parallelism grows in coming years. In the long run, we expect that to some extent, it will remain beneficial to distribute and load-balance work queues across multiple processors to reduce contention. However, a hybrid structure might be the best answer in the future.

Our work is the first to combine a work-stealing approach for task queue management with uberkernel and persistent-thread programming styles to exploit task parallelism and ensure efficient scheduling for irregular work queuing. Together these techniques can indeed effectively leverage the large compute and bandwidth capabilities of the modern GPU, while mitigating the difficulties of its more restrictive programming model. In the future we hope to explore, and hope others will explore, further abstractions and models for continuing to improve the efficiency of the development and the execution of programmable pipelines.

Acknowledgments

We would like to thank Matt Pharr, Aaron Lefohn, and Mike Houston for their suggestions in the early stages of the work; the anonymous reviewers for their excellent reviews; Jonathan Ragan-Kelley for his thoughtful comments on revising the final version; and Shubho Sengupta and Aaron Lefohn for reading early drafts of the paper. Thanks to Bay Raitt for the Big Guy model and Headus Inc. (http://www.headus.com.au) for the Killeroo model. Thanks also to our funding agencies, the National Science Foundation (Award 0541448) and the SciDAC Institute for Ultrascale Visualization, and to NVIDIA for equipment donations and an NVIDIA Graduate Fellowship.

References

[ABP98] ARORA N. S., BLUMOFE R. D., PLAXTON C. G.: Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures* (June/July 1998), pp. 119–129. 3

[AL09] AILA T., LAINE S.: Understanding the efficiency of ray traversal on GPUs. In *Proceedings of High Performance Graphics 2009* (Aug. 2009), pp. 145–149. 2, 3

[BL99] BLUMOFE R. D., LEISERSON C. E.: Scheduling multithreaded computations by work stealing. *Journal of the ACM 46*, 5 (Sept. 1999), 720–748. 2, 5

[CCC87] COOK R. L., CARPENTER L., CATMULL E.: The Reyes image rendering architecture. In *Computer Graphics (Proceedings of SIGGRAPH 87)* (July 1987), pp. 95–102. 2, 5

[CJvdP07] CHAPMAN B., JOST G., VAN DER PAS R.: *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007. 2

[CT08] CEDERMAN D., TSIGAS P.: On dynamic load-balancing on graphics processors. In *Graphics Hardware 2008* (June 2008), pp. 57–64. 3

[EL10] EISENACHER C., LOOP C.: Data-parallel micropolygon rasterization. In *Eurographics Proceedings* (2010). 3

[FFB*09] FISHER M., FATAHALIAN K., BOULOS S., AKELEY K., MARK W. R., HANRAHAN P.: DiagSplit: Parallel, crack-free, adaptive tessellation for micropolygon rendering. *ACM Transactions on Graphics 28*, 5 (Dec. 2009), 150:1–150:10. 3, 8

[FLB*09] FATAHALIAN K., LUONG E., BOULOS S., AKELEY K., MARK W. R., HANRAHAN P.: Data-parallel rasterization of micropolygons with defocus and motion blur. In *Proceedings of High Performance Graphics 2009* (Aug. 2009), pp. 59–68. 3

[HA98] HEIRICH A., ARVO J.: A competitive analysis of load balancing strategies for parallel ray tracing. *The Journal of Supercomputing 12*, 1-2 (Jan. 1998), 57–68. 2

[HS08] HERLIHY M., SHAVIT N.: *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008. 2

[LHLW10] LIU F., HUANG M.-C., LIU X.-H., WU E.-H.: FreePipe: a programmable parallel rendering architecture for efficient multi-fragment effects. In *I3D '10: Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (Feb. 2010), pp. 75–82. 1

[PBD*10] PARKER S. G., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A., STICH M.: OptiX: A general purpose ray tracing engine. *ACM Transactions on Graphics 29*, 3 (Aug. 2010). 2

[PEO09] PATNEY A., EBEIDA M. S., OWENS J. D.: Parallel view-dependent tessellation of Catmull-Clark subdivision surfaces. In *Proceedings of High Performance Graphics 2009* (Aug. 2009), pp. 99–108. 2, 3

[Pha06] PHARR M.: Interactive rendering in the post-GPU era. Keynote, Graphics Hardware 2006. http://www.graphicshardware.org/previous/www_2006/presentations/pharr-keynote-gh06.pdf, Sept. 2006. 1

[PO08] PATNEY A., OWENS J. D.: Real-time Reyes-style adaptive surface subdivision. *ACM Transactions on Graphics 27*, 5 (Dec. 2008), 143:1–143:8. 3

[SCS*08] SEILER L., CARMEAN D., SPRANGLE E., FORSYTH T., ABRASH M., DUBEY P., JUNKINS S., LAKE A., SUGERMAN J., CAVIN R., ESPASA R., GROCHOWSKI E., JUAN T., HANRAHAN P.: Larrabee: A many-core x86 architecture for visual computing. *ACM Transactions on Graphics 27*, 3 (Aug. 2008), 18:1–18:15. 1

[SFB*09] SUGERMAN J., FATAHALIAN K., BOULOS S., AKELEY K., HANRAHAN P.: GRAMPS: A programming model for graphics pipelines. *ACM Transactions on Graphics 28*, 1 (Jan. 2009), 4:1–4:11. 3

[TK09] TATARINOV A., KHARLAMOV A.: Alternative rendering pipelines using NVIDIA CUDA. Talk at SIGGRAPH 2009, http://developer.nvidia.com/object/siggraph-2009, Aug. 2009. 4

[ZHR*09] ZHOU K., HOU Q., REN Z., GONG M., SUN X., GUO B.: RenderAnts: Interactive Reyes rendering on GPUs. *ACM Transactions on Graphics 28*, 5 (Dec. 2009), 155:1–155:11. 1, 2, 3