

A GPU Task-Parallel Model with Dependency Resolution

Stanley Tzeng

Brandon Lloyd

John D. Owens

Abstract

We present a task-parallel programming model for the GPU. Our task model is robust enough to handle irregular workloads that contain dependencies. We present two dependency-aware scheduling schemes—static and dynamic—and analyze their behavior using a synthetic workload. We apply our methods to intra prediction in the H.264 video codec and an N -queens backtracking problem.

Keywords: I.3.1.a Graphics processors; I.3.1.d Parallel processing; I.4.2.e Video coding.

1 Introduction

Current languages for GPUs provide a data-parallel programming model that runs programs on the graphics hardware. A user writes a kernel and specifies the amount of work that the kernel will process on the GPU; the GPU’s internal hardware scheduler determines how to distribute individual *blocks* of the work to the cores (streaming multiprocessors, or SMs). In recent years this model has been successful for a broad class of computationally-demanding, general-purpose applications.

A key component of this programming model is that the blocks that are distributed to GPU cores are *independent*. No block can have a dependency on

another block. This gives the hardware scheduler freedom to efficiently schedule work onto cores. But it also eliminates a broad class of workloads from consideration: *irregular workloads with dependencies*. Irregular workloads produce a variable amount of output per work unit. Dependent workloads have work that must be completed before other portions can begin. Modern GPUs have a limited ability to synchronize and communicate between different work units. Obtaining good performance from a data-parallel model in the presence of dependencies is a challenge. Because many of these irregular workloads with dependencies still exhibit significant parallelism, we hope that massively parallel processors like GPUs could be a good fit for them. Algorithms with complex irregularities, dependencies, or both are often deemed “unsuitable for GPU computing” and are instead often substituted with more regular algorithms that are less efficient.

We believe this is a misconception, and in this article we show a programming model for GPUs that can handle irregular parallelism with dependencies. We break away from the data-parallel perspective of a GPU and build a task-parallel model on GPU hardware. Our goal is to design a programming model that can handle irregular workloads without a major loss in efficiency. We demonstrate a custom queuing system and a custom scheduler to ensure evenly distributed irregular work to all cores. Further, when

there are dependencies between work, our scheduler can respect those dependencies in scheduling work.

2 Alternative Designs

Recently, GPU task-parallel models have been a popular topic in GPU research. Alia and Laine [1] presented the idea of persistent threads for handling irregular ray generation in raytracing and this was further developed in a GPU raytracer OptiX [9]. Cederman and Tsigas [3] analyzed dynamic load balancing on GPUs with work-stealing schemes. Tzeng et al. [11] used distributed queues with a work-donation scheme to implement a Reyes renderer on the GPU. A comprehensive survey of CPU task-parallel techniques can be found in the work by Arora et al. [2].

Our review of the literature in this area is by no means a comprehensive one. We reviewed the most relevant ones to this article. While the references that we just listed presents different strategies for load balancing and alternative tasking designs, what we present in this article is one that is simple enough for readers to grasp and understand. Interested readers are encouraged to follow up on the references presented in this section and pursue further work.

3 Task-Parallel Programming Challenges

Task parallelism is a natural model for expressing dependencies. Task parallelism, as opposed to data parallelism, can have a different execution path per unit of parallel work. While data-parallel formulations often imply uniform, regular workloads, task parallelism makes no such guarantees: the amount of work that can be executed in parallel at any given time is irregular. Tasks allow us to express dependencies on a higher level.

The current GPU programming model, when running task-parallel code, has several major issues that together result in a significant loss of efficiency:

Tasks spawning tasks GPU kernels assume their launch bounds are fixed for the entire span of work available. When tasks generate other tasks, the amount of work that results is unpredictable and irregular. The only solution is to launch *another* kernel to execute on the newly spawned tasks. This causes additional kernel launching overhead and there is no bound on the number of additional kernel launches.

Separate execution paths for tasks The data-parallel model on GPUs prefers small variance and minimal branches in its execution paths. Running two adjacent threads that have different execution paths on the same core results in the sequential processing of both.

Load balancing Tasks need to be evenly distributed across all cores to maximize parallel efficiency. This includes the tasks that are spawned from tasks as well. The hardware scheduler is unaware of tasks and how they should be scheduled. An efficient task scheduler, specialized for task workloads, is necessary for good load balancing.

There are many different abstractions for GPU computing, each with their own terminology. As our implementation is done on NVIDIA hardware, we will be focusing on the CUDA [8] model of hardware abstraction.

4 Conquering the Challenges

Separate execution paths for tasks: On NVIDIA hardware, threads are grouped into 32-thread units

called *warps*. All threads in a warp run in lockstep and in the original data-parallel model, each data element is mapped onto a thread. Rather than mapping a task to an individual thread, we view one warp process as a task. That is, in our launch bounds, the block size is set to 32 threads per block. Since each warp works on a task, we call it a *worker*.

It is important to understand that this action exposes SIMD parallelism. Rather than viewing a warp as 32 individual threads, we now view it as a single thread with a 32-wide vector. Because each warp executes in lockstep with its own instruction counter, our parallelism is now MIMD across warps rather than SIMD across threads.

Load balancing: Current GPU languages do not make the hardware scheduler available as a resource to the programmer. We must implement a software scheduler to ensure that tasks are distributed to the SMs evenly. Our software scheduler consists of a global queue of tasks where workers may enqueue or dequeue tasks from different ends of the queue. A single global queue of tasks allows fair distribution of tasks and ensures that workers are never starved. Queues must maintain coherence and prevent race conditions; we achieve this with locks.

Each queue is guarded by a lock that each worker can grab atomically. We implement our locks as spin-locks, using the atomic compare-and-swap intrinsics (for example: `while(atomicCAS(lock,0,1) == 1)`). Recent GPUs resolve atomics within internal caches, so spin-locks are now relatively fast.

Tasks spawning tasks: Workers must continue to fetch tasks from the queue until all tasks, including new ones spawned from existing tasks, are processed. We use a programming style known as *persistent threads* [1] to handle this. In a traditional GPU style, each thread is spawned, executes, and dies. In the per-

sistent threads style, GPU threads stay alive through the entire kernel, continuing to run inside a loop and fetch work until the queue is empty. Because persistent threads decouple the number of threads launched from the amount of work to process, we are better able to handle irregular workloads.

Putting it all together: Workers consisting of 1 warp dequeue a task from the global queue. Based on the task, the worker runs one of the execution paths in the kernel. This turns a task into an output and potentially an extra task (the task may spawn a new task). The output of the task goes onto an output queue, and the extra generated task is enqueued back onto the global queue. This model, while sufficient for many tasking purposes, lacks any way to determine if a task has dependencies; tasks are assumed to be independent from one another. In the following section we will augment the current model so that it can be dependency-aware.

5 Dependency Resolution

A key observation is that tasks placed in the queue are processed by the next idle worker. Dependencies, therefore, only affect which tasks may be placed in the queue. Our dependency resolution scheme augments each task with more information to ensure that it can be enqueued when its dependencies are satisfied.

A task can be placed in the queue when all of its dependencies have been resolved. We maintain a count of each task's outstanding dependencies *DCounter*. When a worker is done with its task *t*, the worker decrements the dependency counter of *t*'s dependent tasks. A lookup table *D* links a task to other tasks that depend on it. If any of those tasks become ready (their dependency count is now zero), then they are pushed

onto the queue. The task dependencies between each other can be mapped into a DAG that we term the *task map*. We initially enqueue all tasks with a depth of zero. Tasks with the same depth cannot depend on each other and thus can be scheduled simultaneously. Algorithm 1 shows a pseudocode example of the scheduler now with the dependency constraints.

```

Require: Ready Queue  $Q_{in}$ 
Require: Output Queue  $Q_{out}$ 
Require: Dependency Map  $D$ 
Require: Dependency Counter  $DCounter$ 
while  $tasksRemaining > 0$  do
  if  $|Q_{in}| = 0$  then
    {If there are no available tasks, then wait for
     dependencies to get resolved...}
    continue
  else
    Acquire  $Q_{in}.head$  lock
     $task \leftarrow pop(Q_{in}.head)$ 
    Release lock
  end if
  Process  $task$  into  $out$ 
  Go through dependencies.
  for all  $d$  in  $D[task]$  do
    atomicDec( $DCounter[d]$ )
    if  $DCounter[d] == 0$  then
      Acquire  $Q_{in}.head$  lock
       $d \rightarrow push(Q_{in}.head)$ 
      Release lock
    end if
  end for
   $out \rightarrow push(Q_{out}.tail)$ 
  update  $tasksRemaining$ 
end while

```

Algorithm 1: Pseudocode of a single worker with dependency resolution added. We call this scheme *dynamic scheduling* as a task is dynamically assigned at runtime to any idle worker.

This scheduling algorithm assigns a task to any idle worker. We refer to this scheme as *dynamic scheduling*. The advantage of dynamic scheduling is its generality: it can properly evaluate data-dependent problems with irregular input and output. Deadlocks cannot occur because tasks are not placed in the queue until they are ready to run. The queuing mechanism needed to implement dynamic scheduling adds some overhead, but this is required to handle workloads that have irregular input. However, for some problems, we may already know the number of tasks (i.e. regular input and output), but cannot execute the tasks due to dependencies. In these scenarios, we can map out which workers will run which task, and schedule the work in a predetermined way. We refer to this scheme as *static scheduling*.

Static Scheduling

If dynamic scheduling is a general dependency-resolution scheme, then static scheduling is one with a more narrow scope. It can only be used for problems in which all the tasks and their dependencies are known beforehand. We assign workers to tasks based on the thread block index. Because not all workers will be scheduled on the GPU at the same time, it is possible for the static scheduler to deadlock if the scheduled tasks depend on a task that has not yet started. To avoid deadlock, the *task map* must be constructed in such a way that no task will be started before any task on which it depends. If we assume that the workers are scheduled onto the GPU in order, we avoid deadlock. On the GPU we use in our experiments, this assumption holds, which we can easily verify by outputting the time at which a worker is first scheduled on the GPU.

Static scheduling's main strength is that it can use the internal GPU scheduler to execute workers in the correct order. It does not need a dynamic queuing system with its associated overheads. The result is

simpler, more manageable code. The downside is that it requires careful programming to be deadlock-free. Further, workers may be forced to idly occupy the GPU’s resources while they wait for their dependencies to be resolved. This will create small periods of idle waiting, or “bubbles”, in between task processing.

6 Case Study Applications

We now present two applications that uses our described tasking system. The first is a GPU video encoding application and the second is an N -Queens backtracking problem on the GPU. Through the two applications we demonstrate how our tasking system and dependency resolution scheme can be applied to implement their respective algorithms on the GPU.

H.264 Intra Prediction

H.264 prediction comes in two formats: inter and intra prediction. They aim to encode using redundancy in the temporal and spatial domains of the video respectively. It is straightforward to take advantage of GPU data parallelism in inter prediction, but intra prediction has proven difficult to parallelize efficiently due to its dependencies. We give only a brief overview of the algorithm and direct the interested reader to the official specification [6] and an excellent book [10] for more details.

Intra prediction seeks to encode a frame as efficiently as possible by exploiting spatial locality. Each macroblock uses pixels from its neighbors that have already been encoded to predict its own pixels. The job of the encoder is to pick the mode that results in the smallest encoding, including decomposing 16×16 macroblocks into 4×4 sub-blocks.

A collection of macroblocks is called a *slice*. Macroblocks within a slice only depend on each other.

They cannot have dependencies from macroblocks from a different slice. A frame may consist of multiple slices. Slices can be used for several reasons, but of particular interest to us, slices can be used to increase parallelism because each slice is encoded independently, i.e. each slice can be seen as a mutually exclusive DAG (see Figure 1).

Implementation Each 16×16 macroblock is a task in our system. This results in a diagonal-major ordering of the tasks. The workers compute scores for all the prediction modes and choose the best one. They also compute scores for all 4×4 sub-blocks. After choosing the best mode for a sub-block, we must encode the sub-block and reconstruct it before moving on to the next sub-block. All of these steps present opportunities to exploit data parallelism. We take advantage of the parallelism within a warp by evaluating multiple sub-blocks simultaneously. Since the size of our workers is 32 threads, we choose to process two 4×4 subblocks at once by assigning a thread to each element. Figure 1 shows the encoding order of sub-blocks within a macroblock. The task then chooses whether to use the 16×16 or 4×4 block size, performs the encoding, and reconstructs the macroblock. As the number of macroblocks is known beforehand, both static and dynamic schedulers can be used for intra prediction.

N -Queens Backtracking

Overview The classical N -Queens is a constraint satisfaction problem of placing N queens on a $N \times N$ chessboard so that none of the queens is in a line of sight with another queen [5]. The problem is solved via backtracking, a technique that incrementally finds a solution by building candidate solution “states” that do not violate the rules of the solution. As soon as a state is found that violates the rule, that state and all its successors are abandoned.

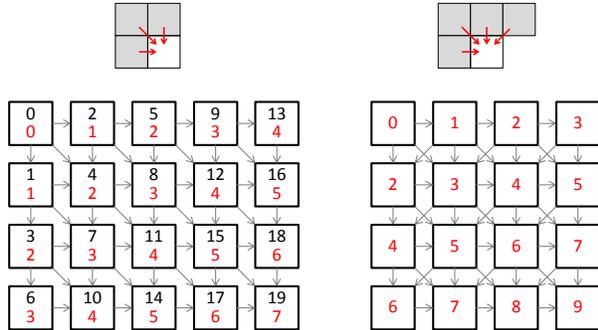


Figure 1: Top row: Dependencies for a 16×16 macroblock (right) and a 4×4 sub-block (left). Bottom row: The task graphs for the 16×16 macroblocks of an 80×48 image (right) and the sub-blocks of a single macro block (left). The red numbers show the maximum depth of a task’s dependency chain. Tasks with the same depth have no mutual dependencies and can be processed in parallel. Black numbers indicate thread block to task assignments. Macroblocks are processed in diagonal major order.

The N -Queens solver builds a tree of the search space where each node represents a partially constructed solution. At each node, the solver attempts to add a new queen to the next unfilled row of the chessboard. If it succeeds, N new nodes are spawned as children of the current node. A solution is found when the entire board is filled. The depth of the tree represents how many rows are filled with queens.

Implementation The N -Queens problem is one where the search space is massive. In order to solve the N -Queens problem in a reasonable amount of time, we only evaluate a board state if we know that its predecessor was valid. In other words, we are pruning the search tree and searching only the valid candidates until we find a solution. Thus, a board state cannot be executed until its predecessor is determined to be valid. This poses a dependency constraint.

We model each tree node as a task. A worker fetches a task, evaluates the state, and determines if the state is valid. If so, it then generates N nodes as potential state candidates and pushes them onto the queue. Otherwise, it simply discards the current task and fetches another. The design of our dynamic scheduler matches nicely to this problem, as we were able to implement this program with relative ease.

This workload would be unsuitable for the static scheduler. The reasons are twofold: first, there are simply too many states; launching a thread block per task is wasteful of resources. Second, since the search tree is pruned, not every task is guaranteed to be executed. Under the static scheduler, this implies that many tasks that are taking up GPU resources may never get executed due to pruning. These two reasons combined make the backtracking problem a more suitable fit for the dynamic scheduler.

The N -Queens problem is a challenging problem for GPU-like architectures due to the combined effect of the large number of dependencies and the disproportionately high memory bandwidth demand. It is for this reason that this style of problem has not been closely studied in the GPU computing literature before. The closest work is the work done by Jenkins et al. [7], whose results on backtracking (with an implementation split between the CPU and GPU) indicate that problems like these are poorly-suited for GPU architectures.

7 Results

Synthetic H.264 Workload

To analyze the performance characteristics of our scheduling schemes, we use a synthetic workload that has the same dependency structure as H.264. We replace the H.264 intra prediction routines with a simple function ($x = \cos(x)$) that is computed for a specified number of iterations. This allows us to concentrate

on only the scheduling component of our system. To visualize the behavior of the scheduling algorithms, we ran one task per streaming multiprocessor (SM) (by allocating all the available shared memory in an SM to a block) and logged timestamps (taken with the `clock()` function) at the beginning and end of task execution. We also log the SM ID (obtained using inline assembly). We used the task graph corresponding to that used for intra prediction of a large task size and a small task size: 720p video frame (1280×720 pixels, 80×45 macroblocks) and a 360p video frame (640×360 pixels, 40×23 macroblocks). We vary task lengths by scaling a base iteration count (500) by a random multiplier in a specified range. Reported timings are the average value over multiple runs.

Figure 2 shows a visualization of the execution of variable length tasks for 3 scheduling techniques and 2 different levels of slice parallelism. With only one initial slice, we see significant underutilization at the start and end of execution where task dependencies restrict the available parallelism. Adding additional slices improves the utilization. With a static schedule, numerous bubbles are visible where workers must wait until the dependencies of their assigned tasks have been resolved. With the dynamic scheduler, bubbles are significantly reduced because workers can be assigned to *any* ready task. Bubbles only occur when there is high contention for the lock on the queue or when no tasks are ready. The performance advantage of dynamic scheduling over static scheduling vanishes when the problem size increases (see Figure 3), especially if the variance in task length is low. This happens because with a large problem size, it takes longer for workers to cycle back to the same region on the front of ready tasks. Thus the probability of being assigned to a task that depends on another task that is still running decreases. If the task length is uniform, then that probability is even slimmer.

Figure 4 demonstrates the runtime for a varying

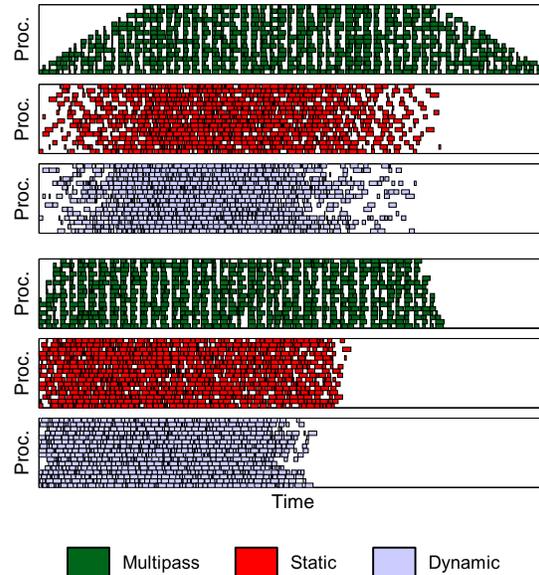


Figure 2: Parallel timelines of variable-length task execution with 15 SMs on a 40×23 graph for a single slice (Rows 1–3). (Row 1) Using global synchronization, the kernel is invoked once for each wavefront. (Row 2) Static scheduling uses finer-grained inter-task synchronization resulting in better utilization. (Row 3) Dynamic scheduling reduces bubbles that can occur when tasks must wait for other tasks to finish. Increasing the number of slices to 4 (Rows 4–6) increases parallelism and improves utilization during the ramp up and wind down stages.

number of slices. We obtain a maximum of 98% improvement for random task lengths and 55% improvement for constant task lengths. Most of the speedup is obtained with just a few initial tasks. The graphs show the relative performance advantage of the dynamic scheduler depends on the size of the problem, as discussed previously. The length of the task is also important because with short tasks, a larger portion of

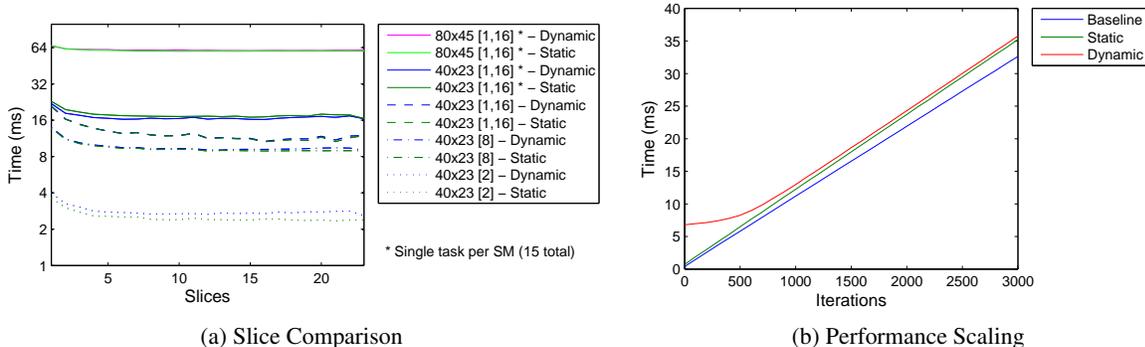


Figure 4: Left: Comparison of performance for varying graph sizes, task lengths, and scheduling algorithms. If we limit the number of SMs to 15, then for large task graphs, slicing makes little difference. A 80×45 graph with random task time multipliers in the range $[1, 16]$ have nearly identical performance for both static and dynamic scheduling. Reducing the size to 40×23 , dynamic scheduling performs slightly better than static scheduling. Removing the limit on SMs or giving the task length multiplier a constant value of 8 eliminates the performance advantages of dynamic scheduling. When task times are further reduced by a factor of 4, scheduling overhead begins to dominate and static scheduling is relatively faster. Right: Performance scaling with a 80×45 graph for varying task length and 0 sleep ticks. The baseline performance executes tasks without waiting for dependencies to be resolved. For short tasks, the performance of the dynamic scheduling method is dominated by locking overhead.

the runtime is lost to scheduling overhead. Figure 4 shows how the performance scales with task length.

H.264 Intra Prediction

Our intra prediction routine is tested against real encoders and videos to see how our dependency resolution scheme fares with realistic scenarios. We examine the encoding time against the open-source, multi-threaded, SSE-optimized x.264 CPU encoder for intra prediction. We also examine how the schedulers deal with intra prediction slices. We are looking to compare with an intra prediction routine that can handle all prediction modes in one pass, as this is most similar to our implementation. Thus we chose to

compare with x.264 rather than the multi-pass method of Cheung et al. [4]. We set the x.264 encoder to the baseline mode so that it most closely resembles the intra prediction routine that we implemented. We ran our tests on a Intel Core 2 Duo E8400 CPU and a NVIDIA GTX 480 GPU running CUDA 3.2.

We chose two video sequences of different resolutions that match the task-graphs chosen for the synthetic scheduler. The first video was one of a bear in a stream with a video resolution of 720×480 pixels per frame. The second was one of ducks flying away from a pool of water. This had a video resolution of 1280×720 pixels per frame.

Table 1 shows the results of our H.264 intra prediction results. We varied the number of slices in a

H.264 Encoder			
Bear Video			
# Slices	x264	Intra Static	Intra Dynamic
1	85.3 ms	26.5 ms	20.4 ms
2	75.4 ms	22.1 ms	15.7 ms
4	60.7 ms	20.5 ms	11.4 ms
Duck Video			
# Slices	x264	Intra Static	Intra Dynamic
1	194.0 ms	72.6 ms	74.3 ms
2	159.7 ms	60.3 ms	65.2 ms
4	133.8 ms	54.1 ms	59.4 ms
<i>N</i> -Queens Solver			
<i>N</i>	GPU	CPU	
15	1.7 s	0.9 s	
16	11 s	7 s	
17	77 s	46 s	
18	580 s	366 s	

Table 1: Top: Intra timing results while doubling the number of slices for the bear and duck videos. As we increase the slices, we also increase the amount of parallelism in the dynamic scheduler. The static scheduler does not increase as much due to the bubbles and sleeps as it waits for dependencies to finish. Bottom: Timing results for the *N*-Queens backtracking problem for relevant sizes of *N*. For sizes $N < 15$ all solutions were found within a second. For sizes $N > 19$, the problem was impractical to execute as the time took too long. However, from the table here we see that our GPU solver comes within 50–80% of the CPU solver. The problem definition is ill-suited for GPU computing due to its branching factor and granularity. Nonetheless, we were able to demonstrate how such a backtracking problem can be implemented using our scheduler.

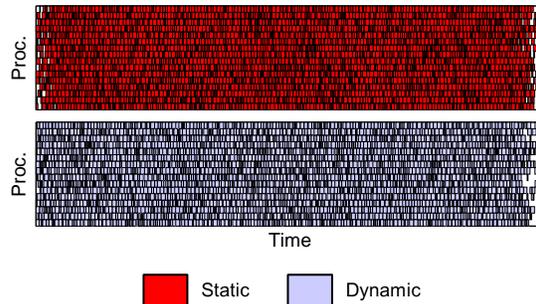


Figure 3: Parallel timelines of variable-length task execution with 15 SMs on a 80×45 graph. (Top) Static scheduling with 4 initial tasks has few bubbles. (Bottom) Dynamic scheduling with 4 initial tasks finishes at nearly the same time as static scheduling.

frame to see how it would affect the runtime. We tried up to four slices, as the majority of the speedup is obtained with the first few slices. Both scheduler versions outperform the CPU version. The dynamic scheduler outperforms the static in the bear scene for two reasons: the number of macroblocks and less idle time. However, when the number of macroblocks are increased in the duck scene, the two are nearly equal to one another, with the static scheduler barely edging past the dynamic scheduler. This matches what we see in the synthetic test results: with a medium task size, the dynamic scheduler is more efficient at scheduling tasks than the static scheduler, but as task sizes grow, the static scheduler is slightly better due to less scheduling overhead.

N Queens Backtracking

We tested our *N*-Queens implementation against a multithreaded SSE-optimized CPU implementation. We measure the time it takes for both versions to find all solutions for a given board size *N*. In order to

record the timings on the GPU, we have manually disabled the watchdog timer so that we could record the timings. We recorded timings up to a reasonable size of N to show how our method compares to the CPU version.

Table 1 shows the results of our GPU N -Queens solver. The branching factor for the N -Queens problem is exponential, as depicted by the difference in runtimes in consecutive values of N . Despite being such a ill-suited problem for GPUs, our naively implemented solver is still able to achieve performance within a factor of two of a highly optimized CPU implementation.

8 Conclusion

In this article we presented a task-parallel programming model capable of handling task-dependencies. We introduced two methods of dependency resolution: dynamic and static. We tested our model by applying it to H.264 intra prediction and N -Queens backtracking.

We separate out the task system into a queue component and a task-dependency components so that they are modular. In the future when there are improved versions of each component, it is trivial to change the scheduler with minor effects to the rest of the system. Further, modularizing the components allows us to quickly analyze results and draw objective comparisons.

Choosing the right combination to use requires a thorough look at the target problem. The static scheduler may be simpler and cleaner, but it requires careful traversal of the task graph to ensure no deadlocks. In the end, there is not one single combination that will outperform all others. Each scheduler requires careful tuning based on its input problem.

Our scheduler as it stands is an early look into the

realm of dependency resolution on GPUs. We expect future hardware generations to support more features that would accelerate our current structure. One important feature would be either a built-in sleep or yield instruction that would put threads to sleep or force threads to give up their resources.

We would like to see more support of robust call stacks and thread preemption in future GPUs. We believe that these hardware feature would directly impact performance and may open up new avenues of exploration in GPU scheduling algorithms.

Acknowledgments

The authors appreciate the support of the National Science Foundation (grants CCF-0644602 and CCF-1017399), the Intel Science and Technology Center for Visual Computing, and a CITRIS seed funding grant.

9 Biography

Stanley Tzeng is currently a PhD student at the University of California, Davis with John Owens as his advisor. His main research interests are task-parallel scheduling, graphics, and GPU computing. He obtained his bachelor and masters at Columbia University. He can be reached via email at stzeng@ucdavis.edu.

Brandon Lloyd received his Ph.D. in Computer Science from the University of North Carolina-Chapel Hill for his work in shadow rendering. His undergrad work was done at Brigham Young University. His interests include computer graphics, parallel programming, and performance optimization. He is currently working in the eXtreme Computing Group (XCG) and can be reached at Brandon.Lloyd@microsoft.com.

John D. Owens is an associate professor of electrical and computer engineering at UC Davis, where he joined the faculty in 2003. His group pursues research problems in GPU computing in both GPU fundamentals and applications. He is a PI and theme leader in the Intel Science and Technology Center for Visual Computing. John received the DOE Early Career Principal Investigator Award in 2004, earned an NVIDIA Faculty Teaching Fellowship in 2006, was the first recipient of his department's Graduate Teaching and Mentorship Award in 2009, and was named an NVIDIA CUDA Fellow in 2012. He graduated from Stanford with a Ph.D. in electrical engineering in 2002 and from Berkeley with a B.S. in EECS in 1995. John is a member of the IEEE. He can be reached via email at jowens@ece.ucdavis.edu.

Bibliography

- [1] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of High Performance Graphics 2009*, pages 145–149, August 2009.
- [2] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, June/July 1998.
- [3] Daniel Cederman and Philippas Tsigas. Dynamic load balancing using work-stealing. In Wen-mei W. Hwu, editor, *GPU Computing Gems*, volume 2, chapter 35, pages 485–499. Morgan Kaufmann, October 2011.
- [4] Ngai-Man Cheung, Oscar C. Au, Man-Cheung Kung, and Xiaopeng Fan. Parallel rate-distortion optimized intra mode decision on multi-core graphics processors using greedy-based encoding orders. In *Proceedings of the 16th IEEE International Conference on Image Processing (ICIP)*, pages 2309–2312, November 2009.
- [5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.
- [6] International Telecommunications Union. ITU-T recommendation H.264 : Advanced video coding for generic audiovisual services, November 2007. <http://www.itu.int/rec/T-REC-H.264-200711-I/en>.
- [7] John Jenkins, Isha Arkatkar, John D. Owens, Alok Choudhary, and Nagiza F. Samatova. Lessons learned from exploring the backtracking paradigm on the GPU. In *Euro-Par 2011: Proceedings of the 17th International European Conference on Parallel and Distributed Computing*, volume 6853 of *Lecture Notes in Computer Science*, pages 425–437. Springer, August/September 2011.
- [8] NVIDIA Corporation. NVIDIA CUDA compute unified device architecture programming guide. <http://developer.nvidia.com/cuda>, January 2007.
- [9] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. OptiX: A general purpose ray tracing engine. *ACM Transactions on Graphics*, 29(3), August 2010.
- [10] Iain E. Richardson. *The H.264 Advanced Video Compression Standard*. Wiley, 2010.

- [11] Stanley Tzeng, Anjul Patney, and John D. Owens. Task management for irregular-parallel workloads on the GPU. In *Proceedings of High Performance Graphics 2010*, pages 29–37, June 2010.