L14: Dynamic Scheduling

Administrative

- STRSM due March 17 (EXTENDED)
- Midterm coming
  - In class April 4, open notes
  - Review notes, readings and review lecture (before break)
  - Will post prior exams
- Design Review
  - Intermediate assessment of progress on project, oral and short
  - Tentatively April 11 and 13
- Final projects
  - Poster session, April 27 (dry run April 25)
  - Final report, May 4

Design Reviews

- Goal is to see a solid plan for each project and make sure projects are on track
  - Plan to evolve project so that results guaranteed
  - Show at least one thing is working
  - How work is being divided among team members
- Major suggestions from proposals
  - Project complexity – break it down into smaller chunks with evolutionary strategy
  - Add references – what has been done before? Known algorithm? GPU implementation?

Design Reviews

- Oral, 10-minute Q&A session (April 13 in class, April 13/14 office hours, or by appointment)
  - Each team member presents one part
  - Team should identify "lead" to present plan
- Three major parts:
  I. Overview
  - Define computation and high-level mapping to GPU
  II. Project Plan
  - The pieces and who is doing what.
  - What is done so far? (Make sure something is working by the design review)
  III. Related Work
  - Prior sequential or parallel algorithms/implementations
  - Prior GPU implementations (or similar computations)
- Submit slides and written document revising proposal that covers these and cleans up anything missing from proposal.

Final Project Presentation

- Dry run on April 25
  - Easels, tape and poster board provided
  - Tape a set of Powerpoint slides to a standard 2'x3' poster, or bring your own poster.
- Poster session during class on April 27
  - Invite your friends, profs who helped you, etc.
- Final Report on Projects due May 4
  - Submit code
  - And written document, roughly 10 pages, based on earlier submission.
  - In addition to original proposal, include
    - Project Plan and How Decomposed (from DR)
    - Description of CUDA implementation
    - Performance Measurement
    - Related Work (from DR)

Let's Talk about Demos

- For some of you, with very visual projects, I encourage you to think about demos for the poster session
- This is not a requirement, just something that would enhance the poster session
- Realistic?
  - I know everyone's laptops are slow …
  - … and don't have enough memory to solve very large problems
- Creative Suggestions?
  - Movies captured from run on larger system

Sources for Today's Lecture

- "On Dynamic Load Balancing on Graphics Processors," D. Cederman and P. Tsigas, Graphics Hardware (2008).

http://www.cs.chalmers.se/~cederman/papers/GPU_Load_Balancing-GH08.pdf

- "A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems," P. Tsigas and Y. Zhang, SPAA 2001.

(more on lock-free queue)

- Thread Building Blocks

http://www.threadingbuildingblocks.org/

(more on task stealing)

Motivation for Next Few Lectures

- Goal is to discuss prior solutions to topics that might be useful to your projects
  - Dynamic scheduling (TODAY)
  - Tree-based algorithms
  - Sorting
  - Combining CUDA and Open GL to display results of computation
  - Combining CUDA with MPI for cluster execution (6-function MPI)
  - Other topics of interest?
- End of semester (week of April 18)
  - CUDA 4
  - Open CL

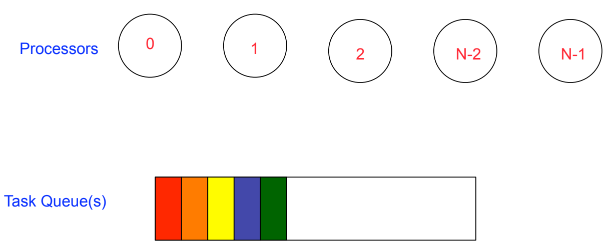
Motivation: Dynamic Task Queue

- Mostly we have talked about how to partition large arrays to perform identical computations on different portions of the arrays
  - Sometimes a little global synchronization is required
- What if the work is very irregular in its structure?
  - May not produce a balanced load
  - Data representation may be sparse
  - Work may be created on GPU in response to prior computation

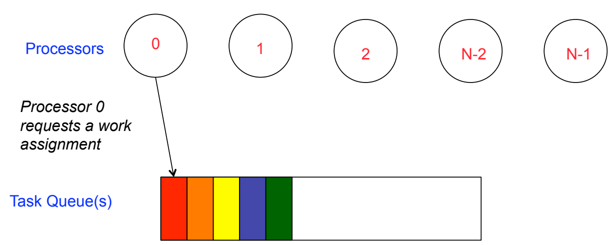
Dynamic Parallel Computations

- These computations do not necessarily map well to a GPU, but they are also hard to do on conventional architectures
  - Overhead associated with making scheduling decisions at run time
  - May create a bottleneck (centralized scheduler? centralized work queue?)
  - Interaction with locality (if computation is performed in arbitrary processor, we may need to move data from one processor to another).
- Typically, there is a tradeoff between how balanced is the load and these other concerns.

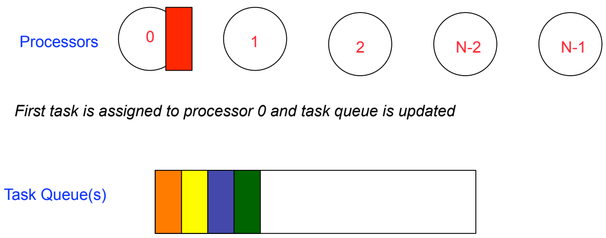
Dynamic Task Queue, Conceptually



Dynamic Task Queue, Conceptually



Dynamic Task Queue, Conceptually



Just to make this work correctly, what has to happen?
Topic of today's lecture!

Constructing a dynamic task queue on GPUs

- Must use some sort of atomic operation for global synchronization to enqueue and dequeue tasks
- Numerous decisions about how to manage task queues
  - One on every SM?
  - A global task queue?
  - The former can be made far more efficient but also more prone to load imbalance
- Many choices of how to do synchronization
  - Optimize for properties of task queue (e.g., very large task queues can use simpler mechanisms)
- All proposed approaches have a statically allocated task list that must be as large as the max number of waiting tasks

Suggested Synchronization Mechanism

// also unsigned int and long long versions

int atomicCAS(int* address, int compare, int val);

reads the 32-bit or 64-bit word old located at the address in global or shared memory, computes (old == compare ? val : old), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old (Compare And Swap). 64-bit words are only supported for global memory.

```
__device__ void getLock(int *lockVarPtr) {
while (atomicCAS(lockVarPtr, 0, 1) == 1);
}
```


Synchronization

- Blocking
  - Uses mutual exclusion to only allow one process at a time to access the object.
- Lockfree
  - Multiple processes can access the object concurrently. At least one operation in a set of concurrent operations finishes in a finite number of its own steps.
- Waitfree
  - Multiple processes can access the object concurrently. Every operation finishes in a finite number of its own steps.

Slide source: Daniel Cederman

Load Balancing Methods

- Blocking Task Queue
- Non-blocking Task Queue
- Task Stealing
- Static Task List

Slide source: Daniel Cederman

Static Task List (Simplest)

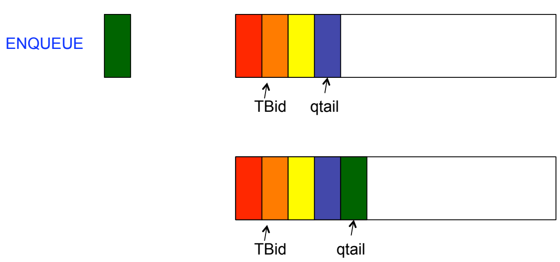
```
function DEQUEUE(q, id)
  return q.in[id] ;
function ENQUEUE(q, task)
  localtail ← atomicAdd (&q.tail, 1)
  q.out[localtail ] = task
function NEWTASKCNT(q)
  q.in, q.out , oldtail , q.tail ← q.out , q.in, q.tail, 0
  return oldtail
procedure MAIN(taskinit)
  q.in, q.out ← newarray(maxsize), newarray(maxsize)
  q.tail ← 0
  enqueue(q, taskinit )
  blockcnt ← newtaskcnt (q)
  while blockcnt != 0 do
    run blockcnt blocks in parallel
      t ← DEQUEUE(q, Tbid++ )
      subtasks ← doWork(t )
      for each nt in subtasks do
        ENQUEUE(q, nt )
    blocks ← NEWTASKCNT (q)
```

Two lists:
q_in is read only and not synchronized
q_out is write only and is updated atomically

When NEWTASKCNT is called at the end of major task scheduling phase, q_in and q_out are swapped

Synchronization required to insert tasks, but at least one gets through (wait free)

Blocking Static Task Queue



Blocking Dynamic Task Queue

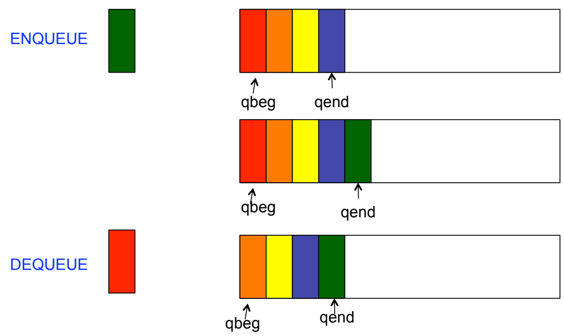
```
function DEQUEUE(q)
  while atomicCAS(&q.lock, 0, 1) == 1 do;
  if q.beg != q.end then
    q.beg ++
    result ← q.data[q.beg]
  else
    result ← NIL
  q.lock ← 0
  return result
function ENQUEUE(q, task)
  while atomicCAS(&q.lock, 0, 1) == 1 do;

  q.end++
  q.data[q.end ] ← task
  q.lock ← 0
```

Use lock for both adding and deleting tasks from the queue.

All other threads block waiting for lock.

Potentially very inefficient, particularly for fine-grained tasks

Blocking Dynamic Task Queue



Lock-free Dynamic Task Queue

```
function DEQUEUE(q)
   oldbeg ← q.beg
   lbeg ← oldbeg
   while task = q.data[lbeg] == NI L do
      lbeg ++
   if atomicCAS(&q.data[lbeg], task, NIL) != task then
      restart
   if lbeg mod x == 0 then
      atomicCAS(&q.beg, oldbeg, lbeg)
   return task
function ENQUEUE(q, task)
   oldend ← q.end
   lend ← oldend
   while q.data[lend] != NIL do
      lend ++
   if atomicCAS(&q.data[lend], NIL, task) != NIL then
      restart
   if lend mod x == 0 then
      atomicCAS(&q.end , oldend, lend )
```

Idea:
At least one thread will succeed to add or remove task from queue

Optimization:
Only update beginning and end with atomicCAS every x elements.

Task Stealing

- No code provided in paper
- Idea:
  - A set of independent task queues.
  - When a task queue becomes empty, it goes out to other task queues to find available work
  - Lots and lots of engineering needed to get this right
  - Best implementions of this in Intel Thread Building Blocks and Cilk

General Issues

- One or multiple task queues?
- Where does task queue reside?
  - If possible, in shared memory
  - Actual tasks can be stored elsewhere, perhaps in global memory

Remainder of Paper

- Octtree partitioning of particle system used as example application
- A comparison of 4 implementations
  - Figures 2 and 3 examine two different GPUs
  - Figures 4 and 5 look at two different particle distributions