L16: Dynamic Task Queues and Synchronization

Administrative

• Midterm
  - In class, open notes
  - Currently scheduled for Wednesday, March 31
  - Some students have requested we move it due to another exam in the previous class
  - Proposed new date, April 5. Vote?

• Project Feedback (early afternoon)

• Design Review
  - Intermediate assessment of progress on project (next slide)
  - Due Monday, April 12, 5PM

• Final projects
  - Poster session, April 28 (dry run April 26)
  - Final report, May 5

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?
  - In some cases, claim no communication but it seems needed to me

• Oral, 10-minute Q&A session (April 14 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 26
  - 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 28
  - Invite your friends, profs who helped you, etc.

• Final Report on Projects due May 5
  - 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)

Sources for Today's Lecture


  (more on lock-free queue)

• Thread Building Blocks
  http://www.threadingbuildingblocks.org/
  (more on task stealing)

Let’s Talk about Demos

• For some of you, with very visual projects, I asked 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

Motivation for Next Few Lectures

• Goal is to discuss prior solutions to topics that might be useful to your projects
  - Dynamic scheduling
  - Overlapping execution on GPU and marshalling inputs
  - Combining CUDA and Open GL to display results of computation
  - Other topics of interest?
  - More case studies

• End of semester (week of April 19)
  - Open GL
  - Future GPU architectures
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

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

Simple Lock Using Atomic Updates
Can you use atomic updates to create a lock variable?
Consider primitives:

```c
int lockVar;
atomicAdd(&lockVar, 1);
atomicAdd(&lockVar, -1);
atomicInc(&lockVar, 1);
atomicDec(&lockVar, 1);
atomicExch(&lockVar, 1);
atomicCAS(&lockVar, 0, 1);
```

Suggested Implementation
```c
// 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 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.

---

**Load Balancing Methods**

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

---

**Static Task List (Recommended)**

**Function Definitions**

```plaintext
function DEQUEUE(q, id)
    return q.in[id];

function ENQUEUE(q, task)
    local tail = atomicAdd(&q.tail, 1)
    q.out[tail] = task

function NEWTASKCNT(q)
    q.in, q.out, oldtail, 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, Tid)
        subtasks = doWork(t)
        for each nt in subtasks do
            enqueue(q, nt)
        blockcnt = newtaskcnt(q)
    end
```

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 Dynamic Task Queue**

**Function Definitions**

```plaintext
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
        end
    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
Lock-free Dynamic Task Queue

function DEQUEUE(q)
oldbeg ← q.beg
lbegin ← oldbeg
while task = q.data[lbegin] == NIL do
    lbegin ++
    if atomicCAS(&q.data[lbegin], task, NIL) != task then
        restart
    if lbegin mod x == 0 then
        atomicCAS(&q.beg, oldbeg, lbegin)
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 work on this is in Intel Thread Building Blocks.

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.
Next Time

- Displaying results of computation from Open GL
- Asynchronous computation/data staging