L19: Dynamic Task Queues and More Synchronization

Administrative

- Design review feedback
  - Sent out yesterday – feel free to ask questions
- Deadline Extended to May 4: Symposium on Application Accelerators in High Performance Computing

  http://www.saahpc.org/
- Final Reports on projects
  - Poster session April 29 with dry run April 27
  - Also, submit written document and software by May 6
  - Invite your friends! I'll invite faculty, NVIDIA, graduate students, application owners, ..
- Industrial Advisory Board meeting on April 29
  - There is a poster session immediately following class
  - I was asked to select a few projects (4-5) to be presented
  - The School will spring for printing costs for your poster!
  - Posters need to be submitted for printing immediately following Monday's class

Final Project Presentation

- Dry run on April 27
  - Easels, tape and poster board provided
  - Tape a set of Powerpoint slides to a standard 2'x3' poster, or bring your own poster.
- Final Report on Projects due May 6
  - 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 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?

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)

Last Time: Simple Lock Using Atomic Updates

Can you use atomic updates to create a lock variable?

Consider primitives:

```
int lockVar;

atomicAdd(&lockVar, 1);
atomicAdd(&lockVar,-1);
```


Suggested Implementation

```
// 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);
}
```


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

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 (Recommended)

```
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 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

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[l beg], 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 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