CS 220: Introduction to Parallel Computing

Introduction to CUDA

Lecture 28

Today's Schedule

- Project 4
- Read-Write Locks
- Introduction to CUDA

Today's Schedule

- **Project 4**
- Read-Write Locks
- Introduction to CUDA

Project 4

- The official spec is posted now
- Extra credit
- Have some fun with it!
  - (if you have time)

Today's Schedule

- Project 4
- **Read-Write Locks**
- Introduction to CUDA

Atomicity

- Our concurrency primitives allow us to ensure **mutual exclusion** in our programs
  - Most common: mutex
- Mutual exclusion helps us prevent race conditions
  - Only one thread is in a critical section at a time
- Our goal here is **atomicity**: making the operations in the critical section appear to be instantaneous

Atomicity - Critical Sections

- When one thread enters a critical section protected by a mutex, other threads have to wait
- While this is happening, the thread can carry out several operations
  - For example: we want to increment three counters
- Once the mutex is unlocked, it **appears** to other threads that all 3 increments happened simultaneously
- This is an **atomic** operation

Linearizability

- Atomic operations **linearize** certain events
  - (make them occur in a particular order)
- This is important in multithreaded environments: we don't control when threads are scheduled
- Related concept: atomicity in database systems
  - When performing a **transaction**, it should appear to happen all at the same time – ***atomically*** – even though multiple operations are taking place
    - Increment bank account, calculate interest, add interest

Read-Write Locks

- The last pthreads concept we will learn is read-write locks:

```
int pthread_rwlock_rdlock(pthread_rwlock_t* rwlock_p);

int pthread_rwlock_wrlock(pthread_rwlock_t* rwlock_p);

int pthread_rwlock_unlock(pthread_rwlock_t* rwlock_p);
```

- What are read-write locks, and why do we need them?

Parallel Linked List

- Let's imagine we're implementing a linked list (sounds familiar, right?)
  - insert
  - delete
  - search
- If we allow concurrent modifications to the list, we could end up with big problems!
  - Insert/delete are not thread safe

List Deletes: Not Atomic

- Suppose Thread 0 is executing **search** and it has loaded the address of the next node
  - `curr_p = curr_p->next_p;`
- Thread 1 races ahead and deletes the next node from the list
- Boom!

Inserts: Also not Atomic!

- Two threads simultaneously execute search and insert
- Thread 0 is executing search and it has loaded the address of the next node
- Thread 1 now inserts a new node after the node Thread 0 is referencing, ***and*** the new node has the desired value.
- Thread 0 will "jump" over the new node and report that the value it's searching for is not in the list.

How do we solve this problem?

- One approach: protect the linked list with a mutex
- What are the downsides?
- Maybe that is too broad. Could we use a mutex for each list item instead?

Mutex Per List Node

- Any time the node is accessed, the mutex must first be locked:

```
if (curr_p->next != NULL) {
   pthread_mutex_lock(curr_p->next->mutex);
   pthread_mutex_unlock(curr_p->mutex);
   curr = curr_p->next;
   /* do stuff */
   . . .
} else {  /* We've reached the end of the list */
   pthread_mutex_unlock(curr_p->mutex);
}
```

Mutex Per Node: Downsides (1)

- Note: it is important to acquire the lock on the next node before relinquishing the lock on the current
- This prevents a thread from racing ahead and changing the pointer in the current node
- If the use of our program involves many searches, but relatively few insertions and deletions, locking and unlocking the nodes will be expensive

Mutex Per Node: Downsides (2)

- Another issue: list size
- If the list is long — thousands or millions of nodes — then we'll vastly increase the cost of traversing the list: adding two function calls for each node!
- And one of the calls can block the thread indefinitely!
- There has to be a better way!
  - (What we all say constantly when writing C programs)

Read-Write Locks

- A lock that gives multiple “readers” simultaneous access to the list, but only one “writer” can access the list at a time
- The basic idea is that we want an object similar to a mutex controlling access to the list
- However, instead of a single lock function, we’ll have two lock functions…

Linked List with R-W locks

- First function: lock the list for reading
  - Any thread that wants to only read the list can proceed
- Second function: lock the list for writing
  - Waits until all reads are done, gains exclusive access to the list, and performs the write
  - This is just like a normal mutex

Benchmarking Threaded LL

Implementation	1 Thread	2 Threads	3 Threads	4 Threads
Read-Write Locks	2.48	4.97	4.69	4.71
One Mutex	2.50	5.12	5.04	5.11
Mutex Per Node	12.00	29.60	17.00	12.00

1000 initial entries, 100,000 total list operations:
80% search, 10% insert, 10% delete

Today's Schedule

- Project 4
- Read-Write Locks
- **Introduction to CUDA**

Graphics Processing Units

- In the old days, video cards were simple devices
- Rendering was done on the CPU
- Unfortunately, this is extremely slow
  - Imagine a low resolution computer screen: 800x600 pixels
  - To render one frame, we have to iterate through all the pixels and update them MANY times per second
- GPUs were created for this specific type of task

Rendering a Frame

- To iterate through a rectangle, we need two **for** loops

```
For (width of the display) {
    For (height of the display) {
        Update pixel i, j
    }
}
```

- On a 800x600 screen, this is 480,000 pixels
- Wouldn't it be better to have 480,000 cores that only needed to do one thing?

GPUs vs CPUs

- Naturally, these differences lead to changes in how we think about implementing our programs
- Nvidia has a nice (but simplistic) video from the folks at *Mythbusters* that "compares" the two:
  - https://www.youtube.com/watch?v=-P28LKWTzrI

Enlightening YouTube Comments

neoqueto 3 years ago (edited)

29x38 resolution, color depth of 7 colors, such a remarkable milestone in computer graphics, NVIDIA!!!

REPLY 350

View all 5 replies

Seunghwa Song 2 years ago

For GPU, It takes more time to press the button though.

REPLY 177

View all 2 replies

General Purpose GPU Programming

- Originally, GPUs were designed for a specific task: graphics
  - (surprise!)
- With games pushing the envelope, GPUs became real powerhouses
  - Even literally: these things can consume major energy and put out lots of heat!
- In the early 2000s, we began to see General Purpose Computing on Graphics Processing Units (GPGPU)

Programming GPUs

- In the early days of GPGPU, programming was difficult
- Programmers often had to “trick” graphics APIs to do the work they wanted
  - Direct3D
  - OpenGL
- These APIs are designed specifically for graphics. Trying to bend them towards general computations was quite difficult

GPU Programming APIs

- Eventually, GPGPU became so popular that the graphics card manufacturers began to support it
- Nvidia: **CUDA**
- ATI (now AMD): **Stream**
- Apple/Industry Group: **OpenCL**
  - Now AMD supports OpenCL as well

CUDA

- **CUDA** is the most popular API, and Nvidia currently has a strong lead in scientific GPGPU applications and machine learning
  - However, it's worth noting that many of today's Bitcoin miners use AMD hardware with OpenCL miner apps
- Originally CUDA stood for **Compute Unified Device Architecture**
  - Now it's just… CUDA

Terminology

- GPUs are much less standardized than CPUs
- In this class, we'll be using Nvidia's terminology for GPU concepts
  - In general, OpenCL et al have similar constructs
- Nvidia GPUs consist of one or more "streaming multiprocessors" (SM or SMXs)
  - Each SM has 8 or more cores and a **control unit**

Department Hardware

- Most of the machines in the department have CUDA capable GPUs and the software installed
- As discussed before, we'll be using the **jet** machines
- Once again, we'll need to use a slightly different compiler to build our programs
  - Running them works as usual, though!

Jet Hardware

- 192 CUDA Cores
- 1889 MBytes memory (shared with system)
- GPU Clock rate: 852 MHz
- Max threads per multiprocessor: 2048

CUDA + MPI

- You **can** use CUDA with MPI
- There is even an API that shares both system memory and GPU memory across systems
- We won't be doing this, though
  - One thing at a time is good enough ☺