CSE 160
Lecture 14

Floating Point Arithmetic
Condition Variables

Announcements

- Pick up your regrade exams in office hours today
- All exams will be moved to student affairs on Wednesday

Today's lecture

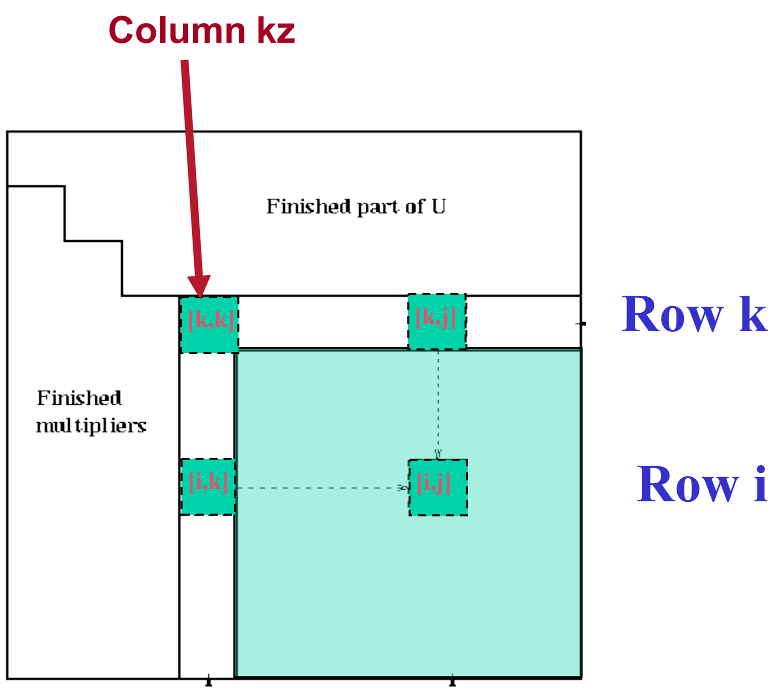
- Revisiting Gaussian Elimination
- Floating Point Arithmetic
- Condition Variables
- Testing and Debugging

Visualizing the Gaussian Elimenation

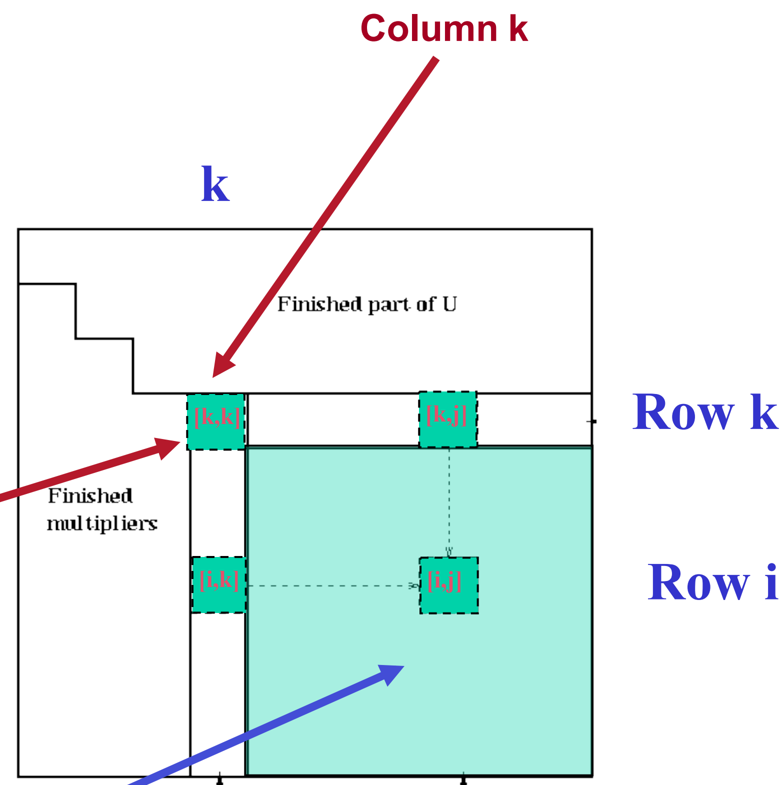
- Add multiples of each row to later rows to make A upper triangular

… for each column k
… zero it out below the diagonal by adding multiples of row k to later rows

```
for k = 0 to n-1
    … for each row i below row k
    for i = k+1 to n-1
        … add a multiple of row k to row I
        for j = k+1 to n-1
            A[i,j] -=  A[ i , k] / A[k,k] * A[k,j]
```



Eliminating the entries below the diagonal

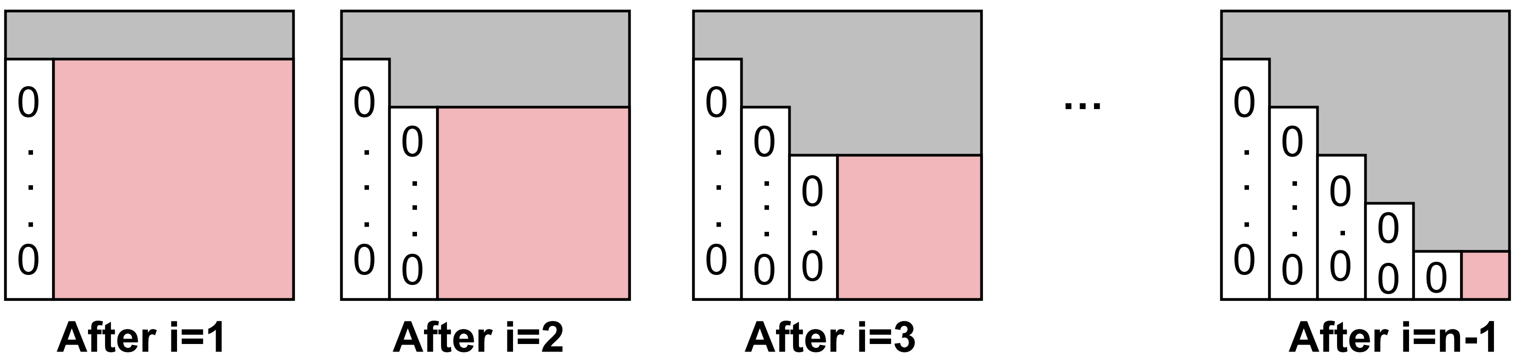


- Add multiples of each row to lower rows to make A upper triangular
- For each column **k : 0 to n-1**
- … subtract multiples of row **k**: **A[k,k+1:n]**
  … from rows i = k+1 to n
- … to zero out column **k** below row **k**
- Multipliers m_{ik} **= A[i,k]/A[k,k]**
- … cancel the elements below the diagonal: **A[k+1:n-1,k]**
- Update only to the right of & below **A[k,k]**

for i = k+1 to n-1
  A[i,k+1:n] – = m_{ik} **× A[k, k+1:n]**

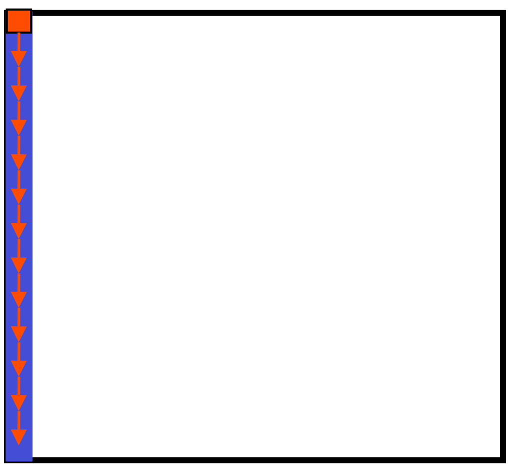
Parallelization

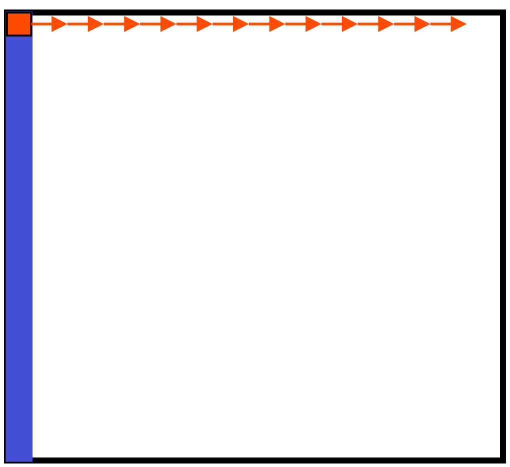
- We'll use 1D vertical strip partitioning
- The ■ represents outstanding work in succeeding k iterations
- We divide the trailing matrix + pivot column among the cores
- The pivot column is always owned by core 0



Communication and control

- Each thread in charge of eliminating N/P columns
- But as we eliminate columns, N is shrinking
- Pivot selection is serial work
- The trailing matrix multiplication parallelizes perfectly





Today's lecture

- Revisiting Gaussian Elimination
- Floating Point Arithmetic
- Condition Variables
- Testing and Debugging

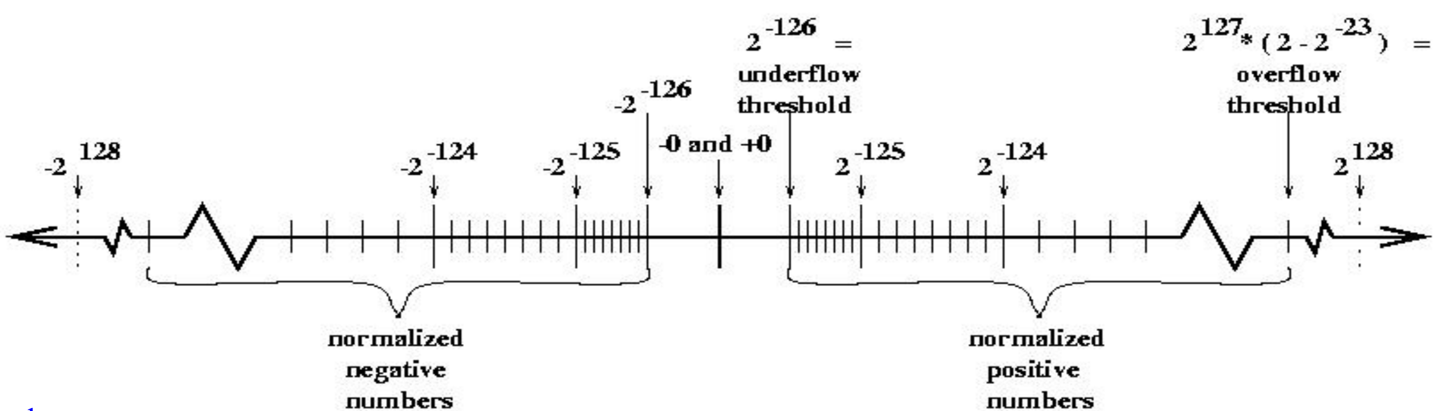
What is floating point?

- A representation
  - $\pm 2.5732\ldots \times 10^{22}$
  - Single, double, extended precision
  - NaN ∞
- A set of operations
  - + = * / √ rem
  - Comparison $< \ \leq \ = \ \neq \ \geq >$
  - Conversions between different formats, binary to decimal
  - Exception handling
- IEEE Floating point standard P754
  - Universally accepted
  - W. Kahan received the Turing Award in 1989 for the design of IEEE Floating Point Standard
  - Revised in 2008

IEEE Floating point standard P754

- Normalized representation $\pm 1.d\ldots d \times 2^{esp}$
  - Macheps = Machine epsilon = $\varepsilon = 2^{-\#\text{significand bits}}$ relative error in each operation
  - OV = overflow threshold = largest number
  - UN = underflow threshold = smallest number
- ±Zero: ±significand and exponent = 0

Format	# bits	#significand bits	macheps	#exponent bits	exponent range
Single	32	23+1	2^{-24} (~10^{-7})	8	2^{-126} - 2^{127} (~10^{+-38})
Double	64	52+1	2^{-53} (~10^{-16})	11	2^{-1022} - 2^{1023} (~10^{+-308})
Double	≥80	≥64	≤2^{-64}(~10^{-19})	≥15	2^{-16382} - 2^{16383} (~10^{+-4932})



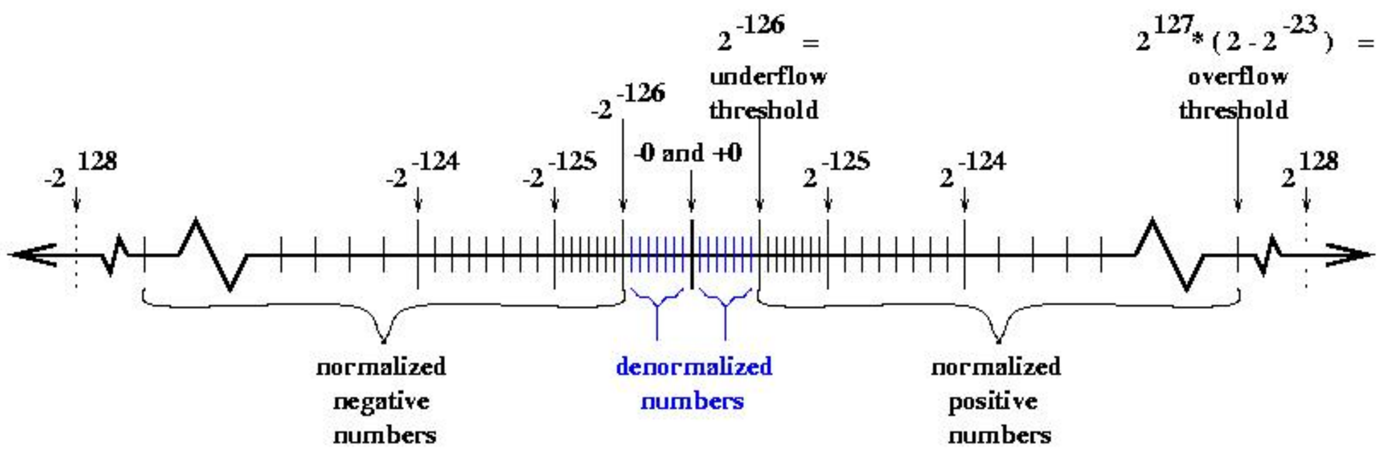
Jim Demmel

What happens in a floating point operation?

- Round to the nearest representable floating point number that corresponds to the exact value(correct rounding)
- Round to nearest value with the lowest order bit =0 (rounding toward nearest even)
- Others are possible
- We don't need the exact value to work this out!
- Applies to + = * / √ rem
- Error formula: $fl(a \text{ op } b) = (a \text{ op } b)*(1 + \delta)$ where
    - op one of + , - , * , /
    - $|\delta| \leq \varepsilon$
    - assuming no overflow, underflow, or divide by zero
- Addition example
    - $fl(\sum x_i) = \sum_{i=1:n} x_i*(1+e_i)$
    - $|e_i| \lesssim (n-1)\varepsilon$

Denormalized numbers

- Compute: if (a ≠ b) then x = a/(a-b)
- We should never divide by 0, even if a-b is tiny
- *Underflow* exception occurs when
  exact result a-b < underflow threshold UN
- Return a *denormalized number* for a-b
  - Relax restriction that leading digit is 1: $\pm 0.d\ldots d \times 2^{min\_exp}$
  - Reserve the smallest exponent value, lose a set of small normalized numbers
  - Fill in the gap between 0 and UN uniform distribution of values



Anomalous behavior

- Floating point arithmetic is not associative

  $(x + y) + z \neq x+ (y+z)$

- Distributive law doesn't always hold

- These expressions have different values when $y \approx z$
  $x*y - x*z \neq x(y-z)$
- Optimizers can't reason about floating point

- If we compute a quantity in extended precision (80 bits) we lose digits when we store to memory $y \neq x$

```
float x, y=…, z=…;
x = y + z;
y=x;
```

NaN (Not a Number)

- Invalid exception
  - Exact result is not a well-defined real number

    $0/0, \sqrt{-1}$

- NaN op number = NaN
- We can have a quiet NaN or an sNan
  - Quiet –does not raise an exception, but propagates a distinguished value
    - E.g. missing data: max(3,NAN) = 3
  - Signaling - generate an exception when accessed
    - Detect uninitialized data

Exception Handling

- An exception occurs when the result of a floating point operation is not representable as a normalized floating point number
  - 1/0, $\sqrt{-1}$
- P754 standardizes how we handle exceptions
  - Overflow: - exact result > OV, too large to represent
  - Underflow: exact result nonzero and < UN, too small to represent
  - Divide-by-zero: nonzero/0
  - Invalid: 0/0, $\sqrt{-1}$, log(0), etc.
  - Inexact: there was a rounding error (common)
- Two possible responses
  - Stop the program, given an error message
  - Tolerate the exception, possibly repairing the error

An example

- Graph the function

$$f(x) = sin(x) / x$$

- f(0) = 1
- But we get a singularity @ x=0: 1/x = ∞
- This is an "accident" in how we represent the function (W. Kahan)
- We *catch* the exception (divide by 0)
- Substitute the value f(0) = 1

Exception handling

- An important part of the standard, 5 exceptions
  - Overflow and Underflow
  - Divide-by-zero
  - Invalid
  - Inexact
- Each of the 5 exceptions manipulates 2 flags
- Sticky flag set by an exception
  - Remains set until explicitly cleared by the user
- Exception flag: should a trap occur?
  - If so, we can enter a trap handler
  - But requires precise interrupts, causes problems on a parallel computer
- We can use exception handling to build faster algorithms
  - Try the faster but "riskier" algorithm
  - Rapidly test for accuracy (possibly with the aid of exception handling)
  - Substitute slower more stable algorithm as needed

Today's lecture

- Revisiting Gaussian Elimination
- Floating Point Arithmetic
- Condition Variables
- Testing and Debugging

Recall Lock_guard

- The lock_guard constructor acquires (locks) the provided lock constructor argument
- When a lock_guard destructor is called, it releases (unlocks) the lock

```
int val, v;
std::mutex valMutex;
...
{
    std::lock_guard<std::mutex> lg(valMutex);
    v = val;
}
    if (v >= 0)
       f(v);
    else
        f(-v);
```

Flexible locking with unique_lock

- The constructor need not acquire (lock) the provided lock constructor argument
- The destructor unlocks the lock, if currently owned

```
std::mutex m;
{
   std::unique_lock<std::mutex> lock_a(m,std::defer_lock);
   std::lock(lock_a);
    ….
} ;
```

Condition variables

- Threads can signal one another with a message: "the buffer is ready"
- Or, we might want to wait for a certain time to elapse: but what if we fell asleep during the alarm?
- We could busy wait on an atomic variable, or a variable protected by a critical section, but this can be wasteful
- C++ provides condition variables, a preferable way to handle the above situations

Using Condition variables

- We need a lock in order to use a condition variable
- Wait() will check for the desired condition within a critical section protected by the lock
  - If the condition has been met, wait() exits
  - Else, it unlocks the mutex and the thread enters a wait state
- Notify_one( ) causes the thread waiting on the condition to awaken
  - The thread reacquires the lock
  - If the condition has been met, Notify_one( ) returns, with the lock in the acquired state
  - Else it blocks again …. How can this happen?

Example with condition variable (I)

```
#include <mutex>
#include <condition_variable>
#include <thread>
#include <queue>

bool more_data_to_prepare() { return false; }
data_chunk prepare_data() { return data_chunk(); }

bool is_last_chunk(data_chunk&) { return true; }

std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;
```

Example with condition variable (II)

```
void Producer_T() {
    while(more_to_produce()) {
        data_chunk const data=Produce();
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(data);
        data_cond.notify_one();
    }
}
void Consumer_T() {
    while(true) {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[]{return !data_queue.empty();});
        data_chunk data=data_queue.front();
        data_queue.pop();
        lk.unlock();
        Consume(data);
        if(is_last_chunk(data))
            break;
    }
}
```

```
int main() {
    std::thread t1(Producer_T);
    std::thread t2(Consumer_T);
    t1.join();
    t2.join();
}
```

Building a barrier with Condition variables

- Cleaner design than with locks only

```
barrier(int NT =2) : ndone(0), _nt(NT){};

  void bsync() {
     unique_lock<mutex> lk(mtx,std::defer_lock);
     lk.lock();
     if (++ndone < _nt)
        cvar.wait(lk);
     else{
        ndone = 0;
        cvar.notify_all();
     }
  };
```

```
Barrier(int NT=2): arrival(UNLOCKED),
    departure(LOCKED), count=0 {};

 void bsync( ){
   arrival.lock( );
   if (++count < NT)  arrival.unlock( );
   else departure.unlock( );
   departure.lock( );
   if (--count > 0) departure.unlock( );
   else arrival.unlock( );
}
```

Today's lecture

- Revisiting Gaussian Elimination
- Floating Point Arithmetic
- Condition Variables
- Testing and Debugging

Fin