CS 33 Marco

marcov@ucla.edu

ssh username@lnxsrv.seas.ucla.edu

scp localfile.c username@lnxsrv.seas.ucla.edu: destination

//this is to copy the file onto some location. The location goes after the colon

1. Download the file

2. Copy the file onto your lnxsrv

CS 33 icecool5280$ scp datalab-handout.tar [calvinl@lnxsrv02.seas.ucla.edu:Desktop/](mailto:calvinl@lnxsrv02.seas.ucla.edu:Desktop/)

**pwd** print working directory. Tells you where you are

**ls** list all the directories

**tab** auto complete

**cd** change directory

**clear** will clear your terminal screen if it gets too full

3. Find the .tar that you saved

**mv** move/rename mv "file" "new name"

4.tar xf datalab-handout.tar //extracts the .tar

5. cd into the unzipped .tar

To check to see if syntax is correct or if you: ./dlc bits.c

If nothing is wrong, then it will say nothing

**make** will show different files

**ls -1** shows you the history of the files

Colors show which files are executables

./ishow 456 will print out all different representation of the number

,.fshow 12.3 shows more values

,.ishow 0x123123 will give you the decimal version as well.

To run and rest everything, run ./btest

If you want to test one function then run ./btest and it will show list of tests that failed and such

Text editor - vim bits.c

**crtl d** is page down

**crtl u** is page up

**capital G** goes to the bottom

**lower case gg** goes to the top

**H** goes to top of screen

**L** goes to the bottom of the screen

**i** is insert mode so you can edit

**esc** brings you back to normal mode

**dw** delete word

**u** undo

**D** deletes everything after cursor

int y = x - 4

**:wq** write and quit

**:q** just quit

**!** after a command overrides are you sure cases

**echo $PATH**

**.** stands for your current directory

**/** next directory

./dlc bits.c will show illegal operator (-) if you use a - (a illegal operaator) in your function.

**:#** will make you jump to a line number

**cd ..** goes back

**/name** will search your doc for that name.

x + ~x always gives you -1 or all 1111's

~x + 1 = -x

11100

111 \* 2^2

(2^3-1)2^2

2^5 - 2^2

If you wanted to multiply by 8 then you would shift something 3 spaces towards the left << 3

If you want to multiply A by 12 then you would:

A \* (2^3 + 2^2)

(A << 3) + (A << 2)

Bitwise Operator

…. means 32 bit

~ Not inverts all the numbers

((~0xA) & 0x3) | 8 = 0….1001

0xA 0….1010

~0xA 1….0101

0x3 0….0011

((~0xA) & 0x3) 0001

8 0….1000

& And

| Or

^ Xor

If both numbers are the same then it is 0, if they are different then the answer is 1

Shift right >> and shift left <<

Dvide by 2 shift to the right by 1.

When shifting to the right the signed bit also gets copied over.

10011…1

After shifting to the right by 3, >> 3 then you get 11110011….1

-8 1000

so dividing it by 2 gives you -4 which is 1100

x&1 is either 0 or 1 to check to see if there is a remainder

x = (y < 0)? a : b;

if y is less than 0, then the result is a

Otherwise it is b

y >> 31

if y >= 0 then 00…..0 (0)

If y <= 0 then 11…..1 (-1) You can now use this with the and operator to keep or kill the numbers

m = y >> 31;

x = (a & m) | (b & ~m);

x is essentially a

! Not

!x if x = 0 then the result is 1

if x is not 0 then the result is 0

!!x if x is 0 then the result is 0

if x is not 0 then the result is 1

reverse the bytes of x

Example: reverse Bytes (0x01020304) = 0x04030201

Legal ops: 1 ~ & ^ | + << >>

Max Ops: 25

Rating: 3

Int reverseBytes (int x) {

//B3 B2 B1 B0

int b3 = (x >> 24) & 0xFF; //will output 0 0 0 B3

}

0xFF = 0……111111111

4/12/13

AND

b & 0 = 0

b & 1 = b

b & b = b

x & 0 = 0

x & -1 = x

x & x = x

OR

b | 0 = b

b | 1 = 1

b | b = b

x | 0 = x

x | -1 = -1

x | x = x

XOR

b ^ 0 = b

b ^ 1 = ~b

b ^ b = 0

x ^ 0 = x

x ^ -1 = ~x

x ^ x = 0

32-bit 2’s complement

0x0…0 0 0

0x0….01 1 1

0x7FFFFFFF 2^31 – 1 Tmax 2^31-1

0x80….0 2^31 Tmin -2^31

0xFFFFFFFF 2^32-1 -1

n bit signed number

[1][][][][][][][][]

n-1

(-2^n-1) it is the negative weight

Sign extend it: [1][1]…..[1][0]

-2^n 2^n-1

-2^n + 2^n-1 = -2^n-1

-x = ~x +1

or (-1 –x) + 1 where -1 is 0xFFFFFFFF

leftBitCount

Ex: Check if upper 16 bits are 1?

mask = (0xFF << 16) | (0xFF << 24)

y = mask & x

z = y ^ mask = 0 if all hi 16 bits are set, else non zero

w = !z

howManyBit

0….0 = 0 1

0….1 = 1 2

1….1 = -1 1

0….010 = 2 3

1…..10 = -2 3

33 – leftbitcount(x) = numberofbitsneeded

condition some mask -> ~0 true result A

* 0 false result B

call this value c (0 ~0)

result = (A & C) | (B & ~C)

IEEE Floating Point

1s\*M\*2E

Encoded [s][exponent][frac] – bit representation

32-bit float exp:8 bits frac:23 bits bias: 127

64-bit double exp: 11 bits frac: 52 bits bias: 1023

exp = 0…0 Denormalized/Zero -> M = 0.frac E = 1-bias

exp = 0…1

exp = 1…10 Normalized -> M = 1.frac (in binary) E = exp – bias = -126 all the way to 127

exp is always unsigned bit pattern

exp = 1….1 NaN / Infinity

Infinity : frac = 0….0

NaN : frac != 0…0

The mantisa has an implied leading 0 and 1, it is not really stored in the bits.

1 3/4

Writing it as binary = (-1)0 \* 1.11 \* 20 exp = 0 +127 from E = exp-127. Exp = 01111111

frac = 110….0 (21 zeros)

Answer: 0x3FE00000

sign exp frac

-2^-130 = (-1)^1 \* 1.0 \* 2^-130 //1.0 because it is already a power of 2

sign | number represented in 2’s | 2^ pwr of something

Cannot have an exponent of -130 because minimum exponent is -126. Must convert it to 2^-126

(-1)^1 \* (0.0001) \* 2^-126, must adjust mantissa because the max cannot be -130

exp = 0…0 because you changed it to denormalized so you use the denormalized equation

frac = 00010…0

Answer: 0x80080000

1000 0000 0000 1000 0..0

1 0111 101

1’s M 2^e

E = exp –bias

M=1.frac

E = 0 because 7-7

Then convert frac into decmical and append it onto the 2^e and then append it to the sign bit

General lnx compiler = **gcc prog.c**

**./a.out to open the output**

**man objdump** can look inside an executable

**a.out | less**

**objdemp –t a.out | less**

**gcc –g prog.c**  is the debugger

**gdb a.out** is like running the debugger

**(gdb) info program**

**(gdb) b main**

**info b**

**run**

**Breakpoint 1, main() at prog.c:5**

howManyBits

ex -5 base 10 = 111 1011 base 2 : r = 4 base 10

(-128 + 64 +32 +16 + 8 + 0 + 2 + 1) = -5

(-8 + 0 +2 + 1) = -5

0100 0001

4/16/13

To put something from remote desktop to local desktop:

Cd into working directory you want to copy into

**Scp** [**calvinl@lnxsrv02.seas.ucla.edu://”path**](mailto:calvinl@lnxsrv02.seas.ucla.edu://) **using pwd”/”filename you want to copy”/\*.txt**

The \*.txt will mean copy all the txt to wherever, needs destination to copy to

**scp calvinl@lnxsrv02.seas.ucla.edu:/u/cs/ugrad/calvinl/Desktop/bomb50/\*.txt .**

**objdump –t bomb >> bombname.txt – will create a txt file on current directory**

**objdump –d bomb >> something.txt – will do the same thing (this is the disassembly)**

**stringsbomb >> bombstrings.txt**

**vi psol.txt = new text editor (it’s just blank)**

**gdb bomb** = will actually enter gdb mode of the bomb or file

**run psol.txt** to actually take in inputs, the code that you found. You do this so it will pass in all the correct things so you don’t need to keep copy and pasting everything

**break** (put function name or line, \*address also works)

Ex) break explode\_bomb //sets breakpoint at beginning of function

break phase\_1 : sets a breakpoint at the beginning of function phase\_1

\*Everytime you lose connection, you have to set breakpoints again\*

**info break** – shows where all breakpoints are

**display/i $pc** //this will show you the next instruction without actually executing.

**display $eax –** will display the eax each time

**display $rsp** – will display the rsp each time

if you see next instruction explode\_bomb, type kill

If there is a completely blank line then it is expecting you to have input

**si** step into

**ni** does not step into the function, it will step over

**n** step through each line in actual code, not assembly

**ni “#”**  where “#” is a number will execute that many assembly line of code

**disas phase\_1** will disassemble phase 1 into each line of assembly code

**x/30c “memory location”** this says print the first 30 characters at that memory location

**x/s “memory address”** this prints out the whole string

4/19/13

Registers –

rax

rbx

rcx

rdx(8 bytes) – edx(4 bytes) – dx(2 bytes)

rsi

rdi

rbp

rsp

r8 …. r15

rip -> inst. Pointer, never really used directly. Tells you where you are in the code, rsp tells you where you are on the stack, but rip is the actual code.

condition flags

Any vriables being declared and arguments will be put on the stack. When you return something, then that info by default susually goes to rax, eax.

-rsp points to the top of the stack.

-[stack]

-[heap]

-[rodata] read only data

-[data] – global viws

-[code] – test

suffix – tells the size of the data you are moving

1 b

2 w

4 l

8 q

mov %rax, %rbx -> rbx = rax

mov %rax,(%rbx) -> \*rbx = rax // rbx will point to rax

movw (%rax), %ax -> reading 2 bytes from rax to ax so it would be short ax = \*rax

mov %rax, 8(%rbx) -> \*(rbx + 8) = rax //the +8 is like accessing array[1] instead of array[8]

Multiplies the offset by the size of your data (64 in this case)

If it was an array of ints then it would be accessing the second element

mov $123, (%rsp, %rbx) -> \*(rsp + rbx) = 123

mov %eax, 4(%rsp, %rbx, 2) -> **(rsp + rbx**2 +4) = eax

lea 4(%eax, %ebx), %ecx -> ecx = (eax + ebx +4)

1. %rdi

2. %rsi

3. %rdx

4. %rcx

5. %r8

6. %r9

7… stack

return val: %rax

hex address Stack

150

148

…

100

rsp holds the address of the top of the stack

**Instructions:**

push %rbp -> rsp -= 8 \*rsp = rbp

pop %rbp -> rbp = \*rsp rsp += 8

150 to 200 because 148 hex + 8 yields 200, has to go through abcdef.

call 0x400ABC->”push %rip + 5” j 0x400ABC

the call instruction is 5 bytes which is why it will return to the instruction pointer and go to the next instruction

ret “pop %rip”

leave

x = foo(3,4);

mov $3, %rdi

mov $4, %rsi

call foo

mov %rax, [x] [x] can be anything, a register exp.

Int Foo(int x, int y)

{

printf(“%d %d”, x, y);

int z = x+y;

return z;

}

**Assembly:**

mov %rsi, %rdx

mov %rdi, %rsi

mov 0x400100, %rdi

call printf

sub $8, %rsp

mov %rsi, (%rsp)

add %rdx, (%rsp)

mov (&rsp), %rax

ret

sub = allocate[8 bytes on stack]

cmp = compare 1 and arg

jg = if edi > 1 then goto Label

eax = 1

then a goto means the return value is 1.

4/26/13

IA -32 Stack Conventions

Command Pseudo

push %ebx esp -= 4

pop %ebx ebx = \*esp esp+=4

call Foo “push %eip+5” //usually func takes 5 bytes, and address is of call instruction

j Foo //to jump into the function

ret “pop %eip”

-cannot hardcode address for ret, need to pop because if you call the function in multiple parts of the code, it will return to that same spot in main and go backwards or forward

-cannot use global variables and return that global variable, need to put the address onto the stack because if you want to use recursion, it won’t work. You need the stack for recursion

Caller-save: ebx, esi, edi //if the system needs to keep these values, it will save and restore it after a function.

Callee-save: eax, ecx, edx //register responsible to save these if they need it and want it there after the function call.

Special: esp, ebp

Caller: (when calling a function)

1. Save eax/ecx/edx (push)

2. push arguments in reverse order (might not even have arguments)

3. execute the “call”

4. save return value somewhere

5. restore any registers saved (eax/ecx/edx) (pop)

6. increment esp (deallocate args) //this is the part where you start going in reverse order on the stack and popping each one off one at a time

Callee:

1. Want to make our own stack frame so you can’t just overwrite ebp because it has the value of the previous function so you need to push %ebp //push and save the old value of ebp

2. set ebp = esp //set it to the current stack pointer for your own stack frame

3. save ebx/esi/edi (push)

4. main function body

-allocate locals

-read args

-do the “real work”

-put the return value in eax to return. Might not even have it is a void

5. set esp = ebp

6. restore any saved registers (pop)

7. restore ebp

8. execute “ret”

lea -4(%ebp), %esp

pop %ebx

is equivalent to

mov %ebp, %esp

mov -4(%ebp),%ebx

int factorial(int x)

{

if (x <= 1) return 1;

return x\* fact(x-1);

}

2

ret address

old ebp

old ebx

x-1 = 1

ret addr

old ebb

old ebx

push %ebp

move %esp, %ebp

push %ebx

mov 8(%ebp),%ebx

cmp $1, %ebx

mov $1, %eax

jle done

.

.

.

done point: pop %ebx

pop %ebp

ret

lea -1(%ebx), %eax //just does ebx-1 and does not access memory

push %eax

call fact

at this point, fact(x-1) will be returned and that will be the new updated version of eax

imul %ebx,%eax //eax = eax \* ebx

add $4, $esp //equivalent to lea 0x4(%esp), %esp

struct Node

{

int value;

struct Node \*next;

}

struct Node\* create\_list(int data[], int N)

{  
 struct Node \*head = 0, \*n;

int i;

for(i = 0; i < N; i++)

{

n = calloc(1, sizeof(\*n));

n->value = data[i];

if(head) head->next = n; //let’s ignore that it is wrong

head = n;

}

return head;

}

N

Data &data[i]:ebx

Ret addr head:esi

Old ebp &data[N]:edi

create\_list:

push %ebp

mov %esp, %ebp

push %ebx

mov 8(%ebp), %ebx //ebx = data

mov $0, %esi //esi = NULL

mov 12($ebp), %edi //edi = N

lea (%ebx, %edi, 4), %edi //edi = data+4N

push $8

push $1 //set up calloc args

Loop:

Call calloc

mov (%ebx), %ecx //load data[i]

mov %ecx, 0(%eax) //n->value = data[i]

test %esi, %esi

je Label //skip next inst if

mov %eax, 4(%esi) //head->next = n

Label:

mov %eax, %esi //head = n

add $4, %ebx //loop inc

cmp %edi, %ebx //loop test

jne Loop

Done:

mov %esi, %eax //return head

move %ebp , %esp

mov -4(%esp), %ebx

pop %ebp

ret

5/3/13

((float)x + f) – f == (float) x No, because if f is really big and x is really small, issue with rounding, f could be infinity even though it is NaN.

(ux\*uy) == (x\*y) Yes it will always be equal because huge number \* hugenumber but if you read just the lower 32 bits, then it will be the same as the right side.

((x&8)|y) == y) No, because we don’t know what the outcome of the left side really is in comparison to the right side.

(x^y)^x+z == y+z Yes, just do it arithmetically -14

1 0111 100

The 8 bit unsigned is 188

An 8 bit two’s complement integer = -68

A floating point- -1.5 -7

j = 48, k = 26

esp points at the return address and then the parameter j and k

The default case can happen before the jump table. The default case starts off after the compare. When you saw 7 being compared to some value and it would return if it was not below, it is saying that it would go to the default.

There is also multiple jump addresses to the default in the print out.

The cin value was 6 -51

links <http://scoreboard> website will display it and q will quit

**objdump –t bufbomb | less**

**objdump –t bufbomb | grep main,** grep is like find

you’re going to be making txt of bytes

bf 66 7b 32 78 /\* mov $0x78327b66,%edi \*/

./hex2raw < sample.txt > sample.bin

**hexdump –C sample.txt //converts the text to bits**

**hexdump –C sample.bin //then coverts it for the input**

**./hex2raw < sample.txt | ./bufbomb –u 804182525**

**run –u 804182525 < sample.bin to run the file**

**./hex2raw –n < sample.txt | ./bufbomb –n –u 804182525**

**if you want to submit the solution: need to attach –s somewhere in ./hex2raw –n < sample.txt | ./bufbomb –n –u 804182525**

**vim sample.s**

**gcc –m32 to compile the file**

**gcc –m32 –c sample.s – wil create object file**

**sometimes it will complain if you push a 4 byte value and don’t put a suffix like l**

**objdump –d sample.o**

**The middle part is what you put in the sample, it is the instruction in hex form.**

**gdb bufbomb**

**b getbuf**

**run and disassemble**

**x/12xw $ebp-0x28 and this will disassemble like the thing on the test**

**you want to overwrite the bytes so it will return to some function that you want it to return to in the memory.**

**Buffer starts at -0x28(%ebp), %eax**

You want to overwrite the return address to be the address of the buffer.

Addresses will be increasing

Address of the buffer itself – you need to track it and find what is being pushed onto the stack.

In nitro stage the buffer will change.

Pushl $0x804….

ret

That push is the address you want it to go to

**cat phase1.txt | ./hex2raw > phase1.raw**

1.

[argument]

[dummy ret] <-esp (fizz)

[ret]

[ebp]

2.Restored ebp in the beginning of test so you can use it as a reference

3.Right before it calls gets it will ppush onto the stack the argument to getS which is the address of the buffer.

[exploit] [padding] [ebp at the beginning of test] [ret to buffer]

[ret to buffer]

[ebp]

[padding]

[exploit]

Practice Test

a. 2^30 [0100]…. [0000] The 1 is in the 30th position

b. 2^30 //if he had testout.x and %u it would still work

c. s = 0 exp = 1000 0000 frac = 0…0

bias = 2^bits -1

E = 1 = exp – bias = 128-127

M = 1.000…

[pos] [1 \* 2^1] [no fraction]

Answer = 2

d. @ prints nothing prints nothing prints nothing

you read in 40 00 00 00 backwards so it is actually printing left to right since it asks for [3] then [2] then [1] ……..

e. 4 bytes because the max is 4 bytes

6.Masking it to get the lower 8 bits, shifting it and then only taking the lower 8 bits

Working backwards

The beginning of the array is some offset + eax\*4 + ebp = eax. You can compute the offset.

Edx does not have an and because movzbl covers that already. -8 because it is index-ebp

-8

Sar is shifting by 8 so it’s $8

And is anding it with 255 so it is $255.

The right side is eax because you can see sar eax. SO the xor line has to be %edx

It is -0x408 because that is the difference between ebp and the hashtable (the start)

-8

-8

$8

$255

%edx

-0x408

Index is a separate variable it is not part of the array/ The hash table is part of the array.

8. You know the arguments must be x y and z. You do not know the order yet though

x86-64 machine

Arg 1 = %rdi

Arg 2 = %rsi

Arg 3 = %rdx

Convention, it is in the slides

Arg 4 = %rcx

Arg 5 = %r8w

Arg 6 = %r9

Arg 7 = rsp+8

Arg 8 = rsp+12

You can see that func 1 has a dx in the first line, that’s 2 bytes, must be a short. The third arg is a short.

The second arg is a long because it uses **r**dx

The 1st arg is an int because of **e**ax

The order is x , y , z because of the return function:

It has to be z \*= 2 because of the add edx edx

It has to be y \*= 18 because rsi+8rsi and then there is an add rsi + rsi

It has to be x \*= 256 because 2^8 with the shift

Func 2 will be just x+y-z because lea rdi+rsi(1st and 2nd arg based off the slides) and then sub edx which is the third argu

The keyword “parallel creates a team of threads. All threads execute the region.

#pragma omp parallel

#pragma omp parallel for

for(\_\_\_;\_\_\_;\_\_\_)

{

}

The number of iterations must be const and known at the loop start. The loop iterations are split among al threads. The first thread will execute loops 0-3 and the second one will do 4-7.

Section A

#pragma omp parallel sections

Section B

The sections are split among all threads.

#pragma omp critical

{

}

All threads will execute, but only one thread at a time. Must be within a parallel region. But eventually they will all be executed.

#pragma omp single

{

}

Only one thread will execute. Other threads wait. Must be within a parallel region.

#pragma omp barrier

No thread will go past the barrier until all threads have reached it. Must be within parallel region.

With “nowait”, the threads do not wait for others to finish.

A variable is shared meaning it has only 1 copy.

If it is private then every thread will have its own copy.

Firstprivate each thread gives each thread an initialized value.

Alignment –

struct P1 { int i; char c; long j; char d; };

biggest thing is long – 8

So it looks like [4][1][align of 3 cuz long doesn’t fit][8][1][align of 7 to fit the 8th check]

structP3{shortw[3];charc[3]};

[2][2][2][1][1][1][align of 1]

struct P4 { short w[3]; char \*c[3] };

w c Total Alignment

0 8 32 8

[return address]

[old ebp]

[something 33-40]

[32]

5/14/13

If there is aliasing,

Load A

Store B

You cannot move store B on top of Load A. If they are not aliasing then you can.

5/17

#pragma omp parallel

{}

The keyword “parallel” creates a team of threads. All threads execute the region.

#pragma omp parallel for

for(\_;\_;\_)

{}

The number of iterations must be constant and known at the loop start. The loop iterations are split among all threads. If some iterations are longer than others, then ou don’t want them to be split evenly because then one has to wait. This is a better balance

#pragma omp parallel sections

#pragma omp critical

{}

All threads will execute but only one thread at a time. Must be within a parallel region.

This is mostly used if you are dealing with shared or global variables.

#pragma omp single

{}

Only one thread will execute, the others wait.

#pragma omp barrier

No thread will go past the barrier until all threads have reached it.

#pragma omp atomic

update-statement

Updates the memory like a++, a[i] = \_\_\_

#pragma omp for nowait

{}

With “nowait”, the threads do not wait for others to finish

What is a thread?

Threads are multiple stacks nd registers.

The lab:

make

make omp

vim main.c

To actually submit:

./submit edgedetect.c to submit the file.

make run ./edgedetect

make check

./clear will clear your submission in progress

./results will show your results

./status shows you how many jobs is being processed

gprof edgedetect | less will give you the time for all functions that it takes.

make GPROF=1 gmon.out will show up.

Lnxsrv uses 16 threads

**[1]#pragma omp parallel for private(variable)**

**[2]#pragma omp for**

**[1]#pragma omp parallel for private(variable)**

**{**

**[2]#pragma omp for**

**for()**

**for()**

**for()**

**}**

**to use pragma omp for, you need to put it into parallel first**

**//compute sum**

**sum = 0;**

**#pragma omp parallel shared(n,a,sum) private(sum\_local)**

**{**

**sum\_local = 0; //each thread will return its own sum local value**

**#pragma omp for**

**for(i = 0; I < n; i++)**

**sum\_local += a[i];**

**#pragma omp ciritical**

**{**

**sum += sum\_local; //global sum**

**}**

**}**

**you want sum\_local to be private because that is what you are returning nd you don’t want that to repeat**

**I is private by default because it is under omp for**

**./submit edgedetect.c**

**./status will tell you when the processing is done**

5/24/13

CUDA overview, Kernel = C function called by CPU

Programmer specifies:

#threads per block

#blocks in grid

Executes on GPU

Each block has “shared memory”

Each device hs global memory

GPU cannot access data from CPU

nvcc : Nvidia C Compiler (similar options to gcc)

**nvcc program.cu –o program**

-deviceemu flag : device emulation. Linux srv doesn’t actually have GPUs so you need to emulate

CUDA coding

*\_Global\_* qualifier : kernel, called by CPU cannot return a value

*\_Device\_* qualifier : called by GPU can return a value

*\_Global\_* void myKernel(…)(…)

*\_device\_* … helperFunc(…)(…)

dim 3 gridDim, blkDim;

gridDim.x = …;

blkDim.x = ....;

Call Kernel using <<<>>> to pass dimensions:

myKernel<<<gridDim,blkDim>>>(…);

In GPU code, get dimensions using:

gridDim(.x, .y .z)

blockDim (.x, .y .z)

blockIdx (.x, .y .z) //block # within grid

threadIdx (.x, .y .z) //thread # within block

Allocte shared memory within a kernel:

\_\_shared\_\_ int myData[16];

Each thread block has its own private copy.

Thread-block barrier within a kernel:

\_\_syncthreads();

All threads within a particular thread block must execute the function before any threads within that thread block can continue

Int val = …;

cudaMemcpyToSymbol (“myGPUGlobal

cudaMalloc(void \*\*ptr, size\_t n\_

cudaFree (void \*ptr)

cudaMemcpy(void \*dst, void \*src, size\_t n, enum cudaMemcpyKind dir)

cudaMemcpyHostToDevice

cudaMemcpyDeviceToHost

cudaMemcpyDeviceToDevice

myKernel<<<…>>>(…) is asynchronous:

Kernel starts, CPU continues on.

cudaMemcpy(…) is synchoronous:

Waits until

\_\_global\_\_void myKernel(int \*A, int c){

int I = blockIdx.x \* blockDim.x + threadIdx.x;

A[i] += c;

}

int main(void) {

int \*A\_host, \*A\_dev;

int N = … , c = … ;

size\_t nbytes = N \* sizeof(int);

A\_host = (int \*) malloc(nybytes);

For(int I =0; I < N; I++) A\_host = …;

cudaMalloc(&A\_dev,nbytes);

cudaMemcpy(A\_dev, A\_host, nbytes cudaMemcpyHostToDevice);

myKernel<<< N/32, 32 >>> (A\_dev, c);

cudaMemcpy(A\_host, A\_dev, nbytes, cudaMemcpyDeviceToHost);

cudaFree(A\_dev);

…

return 0

}

What if N is not a multiple of 32? Then you will create only 1 thread block of size 32

Blk0 Blk1 Blk2

[ ] [ ] [ ]

32 Thread 32 Thread 32 Thread

N = 96

I = 0, 1, 2, 3, ….95

blkDim 32 32 …… 32

threadIdx 0 1 2 3 … | 0, 1, 2, 3…

blkIdx 0 0 0 0 0 0 | 1 1 1 1 1

You multiply the blkIdx with the blkDim and then add the threadIdx

make omp MTRACE=1

make run

make checkmem //check if memory leak

make check

5/31/13

int\* data

int i = blockIdx.x;

int j = threadIdx.x;

data[ I \* blockDim.x + j] = 1000 \* i + j]

**to compile using emulation – nvcc –deviceemu – ccbin /usr/bin myFirstKernel.cu**

**./a.out**

Julia –

Loop:

a = a^2 + c

until a is “infinite”

./a.out

6/6/13

High level concepts OS virtual memory, processes, exceptions, linking

-Multiple Choice 10 points

-Choose one of two 10 points

Floating points, rounding, precision,

CUDA – What does it output, what does it do, in the past: improperly sync threads 10 points

OPENMP – question will be along the times of functional correctness 10 points

MIPS – 10 points

Optimization – understand the Australian animal question, blank fill in 20 points

Stack Smash / Buffer – more difficult, covers more of the first half, not as hard as nitro, understand the stack and the buffer, 30 points

Q2

Where did the dump come from

In the for loop, just left the gets function

What should happen after the call to gets, what does stack look like

&tempbuf -> [Param]

[ret]

[old ebp]

When at the end of the procedure call, esp should be pointing to the ret, and then esp will point to the param

Esp is where the buffer is actually located at ffffdb40

The buffer is at 0x74617257

Understand first lecture of cuda

Example question – Simultaneous Multi Threading

a) increase clock rate

b) increase processor util

c) decrease miss rate

d) …….

Synchr vs. asychron <- happens on the chip

| Exceptions = interrupt, crtl c

|

---🡪 trap – system call

fault – page fault, seg fault

abort - machine check

CISC vs. RISC

Variable length instruction, goal is to save memory space.

The same concept of optimizing with structures where you compact everything since the “falvors” are variable length. Hard to implement

RISC – small number of instructions

Compiler and hardware work together to provide parallelism and piplining

Fix length in terms of instruction (bytes), heavy use of register

Pseudo instruction s

MIPS is RISC

Every instruction in MIPS is 32 bits

Each process has it’s own page table and

For(yy = 0; yy < N; y+=BSIZE)

For(x = 0; x < N x++)]

For(y = yy; y < MIN(N, yy+BSIZE)); y++)

[ x ]

Yy = 0

|

|

v

yy = BSIZE \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|

|

v

yy = BIZEx2\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|

|

v

yy = n

for(xx = 0; xx < N; xx+= BSIZE)

for(yy = 0; yy < N; yy += BSIZE)

for(x =xx; x < MIN(N, xx+BSIZE); x++)

for(y = yy; y < MIN(N, yy+BSIZE); y++)

row \* col

turn into single element multiplied by row = sum row

for(k….

for(i….

r = a[i][k]

for(j….

c[i][j] += r \* b[k][j]

for(jj = 0; jj < n; jj+BSIZE)

for(i = 0; i < n; i++)

for(j = jj; j < min(j+BSIZE,n); j++)

c[i][j] = 0

for(kk = 0; kk < n; kk+=BSIZE)

for(i = 0; i < n; i++)

for(j = jj; j < min(jj+bSIZE, n); j++)

sum = 0;

for(k = kk; k < min(kk+BSIZE,n); k++)

sum += a[i][k] \* b[k][j];

c[i][j] += sum

n might be the width, might be the height depending on the matrix

F = 2 new[0][0] = avg(old[0][0], old[0][1], old[1][0], old[1][1])

for each pixel (i,j) in new []

new[i][j] = 0

for each offset(u,v) in block of size FxF

new[i][j] += old[i \* F + u][j \* F + v]

\*new[i][j] /= F\*F

F is the dim of the new matrix \* the scaling factor

6/7/13

On the final, there are going to be a lot of different functions and you’re going to have to use the address of the called function.

The buffer question is not going to have a memory dump

Call chain slide 7

Study blocked matrix, print out all the slides

PSEUDOCODE FOR ORIGINAL FUNCTION:

(F is shrink\_factor)

for each pixel (I,j) in new[]

new[i][j] = 0

for each offset (u,v) in block of size FxF

new[i][j] += old[i\*F + u][j \*F +v]

new[i][j] /= F\*F

ASSUMPTIONS FOR NEW CODE:

Dim\_new = 100 //size of 1 row in new[]

Shrink\_factor = 4

BSIZE = 10 dimension of 2D block in new[]

Dim shrink = 400 //size of 1 row in old[]

Sf2 = 16 //number of elements in old[] to average

Sf2dim = 16\*100 size of 4 rows in old[]

Bdim = 10\*100 //size of 10 rows in new[]

Ii : 0, 10, 20, …….. //row index of first row in new[] block

Jj : 0, 10, 20 ….. //col index of first col in new[] block

I : ii, ii+1, ii+2 ….. MIN(ii+BSIZE,100) //row index in new[] (A:wallaby)

J : jj, jj+1, jj+2, …. MIN(jj+BSIZE, 100) //col index in new[] (C:dingo)

Iidim : 0, 1000, 2000, offset to row ‘ii’ in new

Wombat : iidim, iidim+100, iidim+200 //offset to row I in new[] (E:wombat)

Platypus : iidim\*16, (iidim+100)\*16, (iidim+200)\*16 .. //offset to row 4 I in old[]

Kangaroo = platypus + 16\*100 //offset to row 4 I + 4 in old[] (B:platypus)

Cassowary : 0, 40, 80, ….. //offset to col 4 jj in an old[] row

Echidna = cassowary+0, cassowary+4 ….. //offset to col 4 j in an old[] row

U : platypus, platypus + 400 … kangaroo //offset to row ‘4 I + {0,1,2,3} in old[]

V : echidna , echidna+ 1…….echidna +4 //offset to col 4 j + {0,1,2,3} in an old[] row

Koala : accumulator of old[] values (D:koala)

**FINAL**

Multiple choice all on OS

Questions like these : Which is better 1 GHZ processor or 4 GHZ processor

Open mp code really long, just fix one of the pragmas

You have cuda code with a main function, it calls its cuda function <<<>>> and then that function does something and the question is what does it print. The problem is you have to check the main for the block size and check your dimensions and then check the dimension of the function and keep in mind that it does it in parallel. (EACH THREAD WILL DO THE PRINT STATEMENT, IF YOU HAVE A FOR LOOP AND THEN THE THREAD THEN IT WILL PRINT MULTIPLE OF THESE)

MIPS QUESTION: What does it print. You have a page code to read. If you understand the 2 example codes that he posted you should be fine. Single value answer

Reading a disk sector, if you want something from disk then it needs to connect to the OS to know what is happening. It sends an interrupt to let the OS know to load something from the disk. Faults are recoverable, page faults are normal, blocking, page getting loaded into memory and the program might not have the data needed so you swap the page.

When you run GCC to link your .o files and then some libraries coming in and the linker picks everything up and turns it into an executable. Know the difference between static linking and non static. Dynamic linking could be even better. If you are just changing one file, you don’t need to recompile everything. An example over dynamic linking.

You need syscalls to jump. Focus on how the sys is called