CS33 – Introduction to Computer Organization

Week 9 Discussion

12/1/2017

Agenda

- Midterm 2 review
- OpenMP lab

Midterm 2 Review

Q1

- (6 minutes) The book explains why a hardware cache should index with the middle bits for the set index, and why it would be a bad idea to index with the high-order bits. Why would it be an even worse idea to index with the low-order bits? Briefly explain.

A1

- (6 minutes) The book explains why a hardware cache should index with the middle bits for the set index, and why it would be a bad idea to index with the high-order bits. Why would it be an even worse idea to index with the low-order bits? Briefly explain.

- Answer: cachelines will be wasted due to duplicates (6pts)
- Rubric
  - 5pts: related to having duplicates entries on cache line
  - 4pts: Inefficient use of cache (without explanation to why ? or with other explanation)
  - 3pts: Decreased spatial locality with reason
  - 2 / 1 pts: Explaining about hazards of high order bits (If only this was part of your answer, I might not have given any pts as that is not what the question is asking.

Q2

- When a process forks, the new process has just one thread, which is a replica of the thread that called the fork function.

- 2a (5 minutes) Can a multiplexed program call fork? If so, briefly explain any cautions you might have for such a program. If not, explain why not.

- 2b (7 minutes) Suppose one thread calls fork at about the same time that another thread is calling exit(0). Briefly describe the race condition that would occur.

A2a

- 2a (5 minutes) Can a multiplexed program call fork? If so, briefly explain any cautions you might have for such a program. If not, explain why not.

- Yes. As long as it takes precaution that the child process does not effect the state of multiplexed program (by doing the work it was forked for).

- Rubric:
  - 5 pts: Yes, and care must be taken to protect multiplexed program flow
  - 4 pts: No and reason is that it might spoil the current program flow or Yes and reason related to the answer
  - 3 pts: Yes and it might slow down multiplexing or related answers.
  - 2 /1 pts: I have tried understanding your logic and figured it might be possible in some very specific case.

A2a

- 2b (7 minutes) Suppose one thread calls fork at about the same time that another thread is calling exit(0). Briefly describe the race condition that would occur.

- If call to fork() happens first, thread will be created while if call to exit() happens, fork would never be called! This is a race condition because the output of the program is not deterministic. - 7pts
- In the question it is given that the threads are not related. Hence there might not be any race conditions at all if they belong to 2 different processes - 7pts
- Moral: Race condition need not occur only because of shared variables.
- Rubric
  - 6/5 pts: Some what related to above answer
  - 4/3 pts: Explaining race condition with some other reason that might be true for specific case.

Q3

- Consider the following C code and its translation to x86-64 assembly language:

```
1  float
2  minus (float x)
3  {
4    union { float f; int I; } u;
5    u.f = x;
6    u.i = -u.i;
7    return u.f;
8  }
```

```
minus:
  movd   %xmm0, %eax
  negl   %eax
  movl   %eax, -4(%rsp)
  movss  -4(%rsp), %xmm0
  ret
```

A3a

- (6 minutes) Suppose we change line 1 to read ‘int’, and line 7 to read ‘return u.i;’. How would the machine code change, and why?

- We get rid of these two lines

```
movl   %eax, -4(%rsp)
movss  -4(%rsp), %xmmo
```

A3b

- (4 minutes) Suppose we change all instances of %eax to %ebx in the assembly code. What would go wrong and why?

- %ebx is a callee-saved register. The caller is expecting this value not to be changed. But this value is changed and will result in some bogus value.

```
minus:
  movd   %xmm0, %eax
  negl   %eax
  movl   %eax, -4(%rsp)
  movss  -4(%rsp), %xmm0
  ret
```

A3c

- (4 minutes) Suppose we change all instances of -4 to 4 in the assembly code. What would go wrong and why?

- We are trashing the return value. The return address is changed and the program may go to wrong location.

```
minus:
  movd   %xmm0, %eax
  negl   %eax
  movl   %eax, -4(%rsp)
  movss  -4(%rsp), %xmm0
  ret
```

A3d

- (8 minutes) What value does ‘minus (-0.0)’ return, from the C programmer’s point of view? Show your work.

- -0.0 is 100000…0
- movd %xmm0, %eax
  -> The same bits as 100000…0 which is INT_MIN
- negl %eax
  -> Bits do not change, even if there is overflow
- The next steps (movl related) is more like taking bits and storing
- So the return value will not change; and value will remain same (i.e. -0.0)

A3e

- (10 minutes) What value does ‘minus (1)’ return, from the C programmer’s point of view? Show your work.

- 1 is 0 01111111 000…0
- movd %xmm0, %eax
  -> %eax is on INT with bits 001111111000…0
- negl %eax
  -> %eax is 110000000111…1 + 1 = 110000001000…0
- Counting the bits to $xmm0 a float:
  1 1000000 000…0
  sign bit is 1 -> negative
  exp 1000001, bias 127 -> 129 - 127 = 2
  M = 0 + 1 = 1
- so return value is -4.0

Q4

- Consider the following C function:

```
void vmul (float *restrict a, float *restrict b, long n) {
  for (long i = 0; i < n; i += 8)
    for (int j = 0; j < 8; j++)
      a[i + j] *= b[i + j];
}
```

- The ‘restrict’ keyword tells GCC that a[i] and b[j] cannot be aliases for the same location, for all i and j. When compiled on lnxsrv07 via ‘gcc –O2 –march=native –c vec.c; objdump –d vec.o’ the output is:

```
0000000000000000 <vmul>:
 0:  test   %rdx, %rdx
 3:  jle    4e <vmul+0x4e>
 5:  sub    $0x1, %rdx
 9:  shr    $0x3, %rdx
 d:  shl    $0x5, %rdx
11:  lea    0x20(%rdi, %rdx, 1), %rdx
16:  nopw   %cs:0x0(%rax, %rax, 1)
20:  xor    %eax, %eax
22:  nopw   0x0(%rax, %rax, 1)
28:  vmovss (%rdi, %rax, 1), %xmm0
2d:  vmulss (%rsi, %rax, 1), %xmm0, %xmm0
32:  vmovss %xmm0, (%rdi, %rax, 1)
37:  add    $0x4, %rax
3b:  cmp    $0x20, %rax
3f:  jne    28 <vmul+0x28>
41:  add    0x20, %rdi
45:  add    0x20, %rsi
49:  cmp    %rdx, %rdi
4c:  jne    20 <vmul+0x20>
4e:  retq
```

When compiled with -O3 instead of -O2, the output is:

```
0000000000000000 <vmul>:
 0:  test    %rdx, %rdx
 3:  jle     53 <vmul+0x53>
 5:  sub     $0x1, %rdx
 9:  xor     %eax, %eax
 b:  shr     $0x3, %rdx
 f:  lea     0x1(%rdx), %rcx
13:  xor     %edx, %edx
15:  nopl    (%rax)
18:  add     $0x1, %rdx
1c:  vmovups (%rdi, %rax, 1), %xmm1
21:  vinsertf128  $0x1, 0x10(%rdi, %rax, 1), %ymm1, %ymm1
28:  vmovups (%rsi, %rax, 1), %xmm0
2e:  vinsertf128  $0x1, 0x10(%rsi, %rax, 1), %ymm0, %ymm0
36:  vmulps  %ymm1, %ymm0, %ymm0
3a:  vmovups %xmm0, (%rdi, %rax, 1)
3f:  vextractf128 $0x1, %ymm0, 0x10(%rdi, %rax, 1)
47:  add     0x20, %rax
4b:  cmp     %rcx, %rdx
4e:  jne     18 <vmul+0x18>
50:  vzeroupper
53:  retq
```

Q4abc

- (8 minutes). Assume that n is a large positive multiple of 8 and assume (unrealistically) that each machine instruction takes 1 clock cycle. Compute the cycle per element (CPE) for the -O2 code. Briefly show your work.

- (8 minutes). Likewise, for the -O3 code.

- (8 minutes). Diagram the data-flow representation of computation for one iteration of the inner loop of the -O2 version. In your diagram, consider each 128- or 256-bit vector to be a single value.

A4a

Initiation overhead

Amortized when
“n is large”

Inner loop

6 instructions
per array element

Outer loop

6 instructions
per 8 array elements

```
 0:   test   %rdx, %rdx
 3:   jle    4e <vmul+0x4e>
 5:   sub    $0x1, %rdx
 9:   shr    $0x3, %rdx
 d:   shl    $0x5, %rdx
11:   lea    0x20(%rdi, %rdx, 1), %rdx
16:   nopw   %cs:0x0(%rax, %rax, 1)
20:   xor    %eax, %eax
22:   nopw   0x0(%rax, %rax, 1)
28:   vmovss (%rdi, %rax, 1), %xmm0
2d:   vmulss (%rsi, %rax, 1), %xmm0, %xmm0
32:   vmovss %xmm0, (%rdi, %rax, 1)
37:   add    $0x4, %rax
3b:   cmp    $0x20, %rax
3f:   jne    28 <vmul+0x28>
41:   add    0x20, %rdi
45:   add    0x20, %rsi
49:   cmp    %rdx, %rdi
4c:   jne    20 <vmul+0x20>
4e:   retq
```

A4a

- Calculating CPE
  - Assume each machine instruction takes 1 clock cycle
  - A jump instruction takes 1 cycle whether taken or not
  - CPE = 6 + 6/8 = 6.75
- Instruction-level parallelism
  - Can we parallelize the inner loop like this?

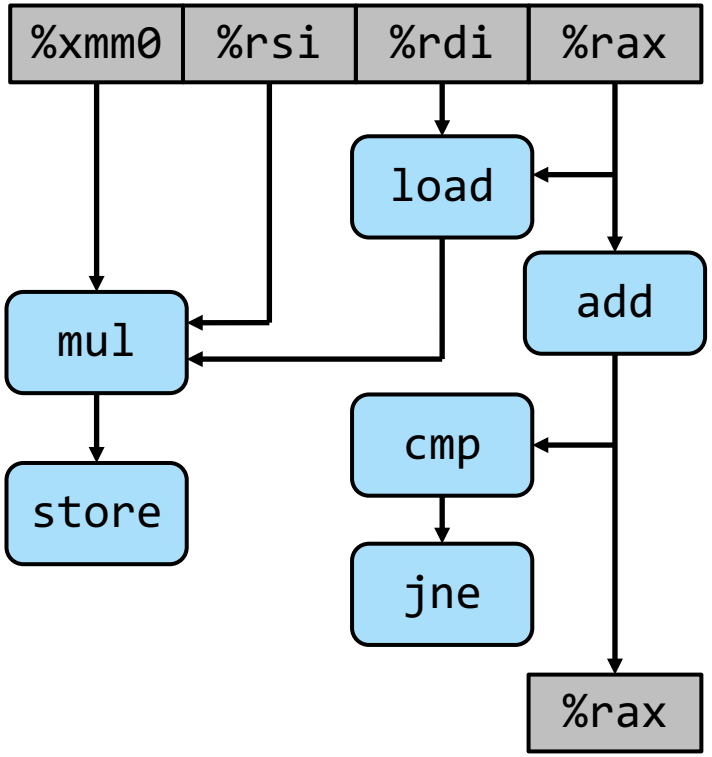
```
vmovss (%rdi, %rax, 1), %xmm0              add    $0x4, %rax
vmulss (%rsi, %rax, 1), %xmm0, %xmm0       cmp    $0x20, %rax
vmovss %xmm0, (%rdi, %rax, 1)              jne    28 <vmul+0x28>
```

  - **WAR dependency can be resolved by register renaming**
  - CPE = 3 + 5/8 = 3.625

A4c

- **"Our data-flow representation of programs is informal. We only want to use it as a way to visualize how the data dependencies in a program dictate its performance."**

```
28:   vmovss (%rdi, %rax, 1), %xmm0
2d:   vmulss (%rsi, %rax, 1), %xmm0, %xmm0
32:   vmovss %xmm0, (%rdi, %rax, 1)
37:   add    $0x4, %rax
3b:   cmp    $0x20, %rax
3f:   jne    28 <vmul+0x28>
```



A4b

Initiation overhead

Amortized when
“n is large”

Loop

11 instructions
per 8 array elements

```
 0:  test    %rdx, %rdx
 3:  jle     53 <vmul+0x53>
 5:  sub     $0x1, %rdx
 9:  xor     %eax, %eax
 b:  shr     $0x3, %rdx
 f:  lea     0x1(%rdx), %rcx
13:  xor     %edx, %edx
15:  nopl    (%rax)
18:  add     $0x1, %rdx
1c:  vmovups (%rdi, %rax, 1), %xmm1
21:  vinsertf128  $0x1, 0x10(%rdi, %rax, 1),
                  %ymm1, %ymm1
28:  vmovups (%rsi, %rax, 1), %xmm0
2e:  vinsertf128  $0x1, 0x10(%rsi, %rax, 1),
                  %ymm0, %ymm0
36:  vmulps  %ymm1, %ymm0, %ymm0
3a:  vmovups %xmm0, (%rdi, %rax, 1)
3f:  vextractf128 $0x1, %ymm0, 0x10(%rdi, %rax, 1)
47:  add     0x20, %rax
4b:  cmp     %rcx, %rdx
4e:  jne     18 <vmul+0x18>
50:  vzeroupper
53:  retq
```

A4b

- Calculating CPE
  - CPE = 11/8 = 1.375
- Instruction-level parallelism

```
add     vmovups         vmovups
        vinsertf128     vinsertf128
        vmulps          add
        vmovups         cmp
        vextractf128    jne
```

  - CPE = 5/8 = 0.625
  - **–O3 does generate faster code**

A4d

- (6 minutes): Explain why ‘restrict’ is important in helping the compiler generate better code for the -03 case.

- Answer: It assumes A and B are not aliases (i.e. not same object, do not point to the same array). If they are not independent, an earlier change of A would create a data dependency.
  - 6 points: discuss aliasing
  - 4 points: discuss overlapping memory

A4e

- (5 minutes): Would the reassociation transformation improve the performance of the vmul function? Explain.

- Answer: No, since vmul does not use the associative property of multiplication here (no accumulation, no need for temporary values, only 2 terms).
  - 5 points: no, since there is no {association, accumulation, temp value}
  - 4 points: no, because floating point multiplication not associative
  - 3 points: no

Q5

- The L3 cache in the Intel Kaby Lake x86-64 processor family is shared across all cores. Depending on the CPU, the L3 cache’s size ranges from 1 to 2 MiB/core, and its set-associativity ranges from 8 to 16-way. Suppose you want to determine the size and associativity of the L3 cache of a Kaby Lake CPU, and you lack access to its serial number of CPU ID so you cannot look it up in a table.

A5a

- (10 minutes): Explain how you could write an “L3 cache inspector program” to infer the size and associativity by having the program time itself.

- Answer: to infer size, create an array and do something that accesses every element in the array (ex. multiplication), and measure performance. Increase size of array until performance drops, and size of array before performance drop should be (close to) size of cache.

  create program that accesses 16 memory “hot spots” in a loop and if performance is not terrible, reduce number. Idea is to purposely cause collisions.

  - +2 points: mention array
  - +3 points: mention increasing size of array
  - +2 points: measure performance drop
  - +2 hot spots/collisions
  - +1 16 hot spots

A5b

- (5 minutes): Would the Kaby Lake's L0, L1, and L2 caches make it easier for you to write your L3 cache inspector program? Briefly explain.

- Answer: No, it would add noise (especially if the associativity is different between the L2 and L3 caches). The L0, L1, L2 caches get in the way of directly measuring the L3 cache.
  - 5 points: No, {adds noise, dependency, etc.}
  - 3 points: No

For further questions, contact

```
1    Sachin
2    Sachin
3    Taqi
4a   Yuchen
4b   Yuchen
4c   Yuchen
4d   Brian
4e   Brian
5    Brian
```

OpenMP Lab

http://web.cs.ucla.edu/classes/fall17/cs33/lab/openmplab.html

OpenMP Lab

- Getting started
  - Download the openmplab.tgz file from CCLE
  - Copy it to lnxsrv07 or 09
  - Unzip it

```
   correct.txt
   filter.c
-> func.c
   func.h
   main.c
   Makefile
   seed.txt
   util.c
   util.h
```

OpenMP Lab

- **To run the sequential code**

```
$ make seq
```

  - **The generated executable is named `seq` (not `filter`)**

```
$ ./seq
FUNC TIME : 0.466126
TOTAL TIME : 2.464838
```

- **Goal: speedup FUNC TIME**
  - **To determine what to speed up: gprof**
  - **To speed it up: OpenMP**

OpenMP Lab

- **To use gprof**
  - `$ make seq GPROF=1`
  - **This generates** `seq`
  - `$ ./seq`
  - **This generates** `gmon.out`
  - `$ gprof seq | less`
  - **Redirects the output as input into the** `less` **command**

OpenMP Lab

- The specs specify methods of invoking the makefile, several of which are wrong
- “make seq” will generate an executable called “seq”
- “make omp” will generate an executable called “omp”
- “make ___ SRC=myfunc.c” does not work
  - If you’d like this functionality to work, in the Makefile change

```
SRC = filter.c
SRCS = $(SRC) main.c func.c util.c
```

  - to

```
SRC = func.c
SRCS = $(SRC) main.c filter.c util.c
```

OpenMP

- OpenMP is an extension to C/C++ (and Fortran) that allows for a simple and convenient way of launching threads to do work in parallel among multiple processors
- You won't be personally using a bunch of new functions
- You'll be specifying a bunch of pre-processor directives
- Preprocessor directives are statements that are marked with "#"
  - `#define min(a, b) (a < b ? a : b)`
  - `#include <stdio.h>`

OpenMP

- **Say you want to parallelize this so that 4 threads run this code**

```
int main()
{
  int a = 0;
  printf(“%d\n”, a);
}
```

OpenMP

- **Say you want to parallelize this so that 4 threads run this code**

```
int main()
{
  omp_set_num_threads(4);
#pragma omp parallel
  {
    int a = 0;
    printf("%d\n", a);
  }
}
```

OpenMP

- **Say you want to parallelize this so that 4 threads run this code**

```
int main()
{
#pragma omp parallel
  {
    int a = 0;
    printf(“%d\n”, a);
  }
}
```

- **If the number of threads is not specified, the default is used. On lnxsrv07 and 09, this means 32.**

OpenMP

- Upon hitting the block specified by the #pragma omp directive, the single thread of execution branches of into several parallel threads
- When reaching the end of the block, each thread will wait for all of the other threads before continuing. This is known as a “barrier”
- This is the basis how OpenMP works

OpenMP

- **Back to this example**

```
int main()
{
#pragma omp parallel
  {
    int a = 0;
    printf(“%d\n”, a);
  }
}
```

- **What if a were declared outside of the parallel section?**

OpenMP

- **Back to this example**

```
int main()
{
  int a = 0;
#pragma omp parallel
  {
    a++;
    printf(“%d\n”, a);
  }
}
```

- **Is a shared among the 32 threads?**

OpenMP

- Data sharing
  - By default, a variable declared inside of a parallel block is private to each block
  - By default, a variable declared outside of a parallel block is shared among all of the threads of a parallel block that accesses that variable
  - A variable within a parallel block can be made private to each thread by using the “private” keyword

OpenMP

- **Warning: the variable will also be private relative to the original declaration**

```
int main()
{
  int a = 123;
#pragma omp parallel private(a)
  {
    a++;
    printf(“%d\n”, a);
  }
}
```

- **The a inside the block will not be initialized to 123. it will be uninitialized**

OpenMP

- If you do want to make the private variable in the block initialized to the value specified in the original thread, use “firstprivate”

```
int main()
{
  int a = 123;
#pragma omp parallel firstprivate(a)
  {
    a++;
    printf(“%d\n”, a);
  }
}
```

- The a inside the block will now be initialized to 123

OpenMP

- **Loops**

```
for (i = 0; i < N; i++) {
  <PARALLELIZABLE CODE>
}
```

- **Use #pragma omp parallel for**

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
  <PARALLELIZABLE CODE>
}
```

OpenMP

- **Accumulation inside a loop**

```
int acc = 0;
#pragma omp parallel for
for (i = 0; i < N; i++) {
  acc += <PARALLELIZABLE CODE>
}
```

- **Problem: acc += is not atomic. The result will not be correct**

OpenMP

- **To ensure that you can use accumulators, use the "reduction" keyword, which allows you to collect results into one shared variable among threads**

```
int acc = 0;
#pragma omp parallel for reduction(+:acc)
for (i = 0; i < N; i++) {
  acc += <PARALLELIZABLE CODE>
}
```

- **Format: (<operation>:<variable>)**

OpenMP Lab

- Final notes
  - The runtime will vary each time you call filter
  - You are supposed to speed up the FUNC TIME as reported by the print statement by 3.5x
  - As it turns out, as the SEASnet processor becomes more burdened, it will actually slow itself down, which will lead to a slower execution time
  - The machine will be the slowest on the due date

OpenMP Lab

- **According to checkmem, OpenMP loves to leak memory**

```
int a;
#pragma omp parallel
{
  a = 0;
}
```

- **Even this leaks memory**

OpenMP Lab

- Memory leaks
  - If the memory error printed out by “make checkmem” is “Memory not freed”, don’t worry about it. Otherwise, go back and double check
  - Make sure to use “make check” to confirm that the output.txt generated by the program is the same as correct.txt. If silent, the files are the same