CS33 Midterm 2

Luke Hyun Jung

TOTAL POINTS

63 / 100

QUESTION 1

1 Q1a **2 / 8**

✓ **- 6 pts** Mostly Incorrect

QUESTION 2

2 Q1b **8 / 8**

✓ **- 0 pts** Correct

QUESTION 3

3 Q2a **1 / 3**

✓ **- 2 pts** Mostly Incorrect, or miss the question

QUESTION 4

4 Q2b **1 / 3**

✓ **- 2 pts** Mostly Incorrect, or miss the question

QUESTION 5

5 Q2c **4 / 5**

✓ **- 1 pts** Mostly correct, some explanation is wrong

QUESTION 6

6 Q2d **4 / 8**

✓ **- 3 pts** Casting instead of interpreting/incorrect return

✓ **- 1 pts** _builtin function

QUESTION 7

7 Q2e **13 / 15**

✓ **- 2 pts** 1 coding error

QUESTION 8

8 Q3 **1 / 6**

✓ **- 5 pts** Mostly Incorrect, or miss the question

QUESTION 9

9 Q4 **2 / 4**

✓ **- 2 pts** reasonable, not the right idea

QUESTION 10

10 Q5 **0 / 6**

✓ **- 6 pts** No answer or incorrect

QUESTION 11

11 Q6a **13 / 16**

✓ **- 3 pts** No or wrong how much faster or wrong of the result or no explain of how much faster or the technique is not the bottleneck, etc.

QUESTION 12

12 Q6b **7 / 8**

✓ **- 1 pts** Almost right

QUESTION 13

13 Q6c **7 / 10**

✓ **- 3 pts** Partially right

UCLA Computer Science 33 (Fall 2018)
Midterm 2
100 minutes, 100 points, open book, open notes.
Write your answers on the exam.

Name: Luke Jung Student ID: 904-982-644

1	2	3	4	5	6	total

1a (8 minutes). Write a C program that accesses an array that runs relatively slowly with the Intel Coffee Lake L2 cache, and which would run significantly faster if the cache were 8-way. As you may recall, this cache is a 256 KiB, 4-way cache with a 64 B line size, and uses a write-back policy. Don't worry about the other caches in the Coffee Lake; assume that they all take zero time.

```
#define N 256

void foo(int dst[N][N], int src[N][N]){
  for (int i=0; i<N; i++)
    for (int j=0; j<N; j++)
      dst[j][i] = src[i][j];
}
```

So for $N \leq 2^7\sqrt{2}$ will be fast.
If $N \geq 2^7\sqrt{2}$ or 2^8, then would be slow for 256 KiB, but fast on 512 KiB

Check Time

256×10^{10}
10^{18}

$4 \cdot N^2 \cdot 2 \leq$ cache
$4 \cdot N^2 \cdot 2 \leq 2^{18}$
$2^3 N^2 \leq 2^{18}$
$N^2 \leq (2^{15}) = 2^{14}2$
$N \leq 2^7\sqrt{2}$

1b (8 minutes). Similarly, write a C program that would run significantly faster if the cache's size were increased to 512 KiB instead.

Answer is above when $N=256$

For the rest of the exam, assume the following for the x86-64 instruction set and for GCC.

* The 'btrq A, B' instruction copies B's bit number A to the carry flag CF, and then sets B's bit number A to zero. A must be in the range 0..63. Bit number 0 is the least significant bit.

* The 'rdrand A' instruction sets A to a random bit-pattern if successful, or to zero if unsuccessful. It also sets the carry flag CF to 1 if successful and to 0 if unsuccessful. rdrand operates by grabbing random data bits from an entropy pool maintained within the chip; normally this should work quickly but if a program tries to grab too many bits too quickly then this will exhaust the pool and rdrand will fail.

* The 'vaddsd A, B, C' instruction sets C to A + B, using double-precision floating-point division.

* The 'vcvtsi2sdq A, B, C' instruction converts the 64-bit signed integer A to a double-precision floating-point number with the same numeric value (rounding in the usual way), and stores the result into the low-order bits of C. The high order bits of C are taken from the corresponding high-order bits of B.

* The 'vdivsd A, B, C' instruction sets C to A / B, using double-precision floating-point division.

* The 'vmovq A, B' and 'vmovsd A, B' instructions copy A to B's low order bits.

* The 'vxorpd A, B, C' instruction sets C to A ^ B.

* GCC has a builtin function with the signature 'unsigned int __builtin_ia32_rdrand64_step (unsigned long long *P);' that uses the rdrand instruction to set *P to a random bit-pattern. The function returns 1 if rdrand was successful and 0 if unsuccessful; when unsuccessful it sets *P to zero.

2. For each of the following functions in x86-64 assembly language, briefly explain what the function does, and write C code suitable for GCC that corresponds to that function. You can define auxiliary functions as necessary, but keep your explanations and your C code as simple as possible.

2a (3 minutes).

```
f1: vmovq  %rdi, %xmm0
    ret
```

```
void vmov(){
    long a;
    long long b = a;
    return;
}
```

Moves a normal 64 bit variable's to all the lower 64 bits of a 128 bit variable/register.

2b (3 minutes).

```
f2: vmovq  %xmm0, %rax
    ret
```

```
{
  long long b;
  long a = b;
  return;
}
```

Sets larger 128 bit register to 64 bit result variable by only shifting lower 64 bits to %rax.

2c (5 minutes):

```
f3: rdrand  %rax
    vmovq   %rax, %xmm0
    ret
```

```
{
  long a;
  int num = _builtin_ia32_rdrand64_step((long long)a);
  long long b = a;
  return;
}
```

function fills an long with random bytes bit pattern and then takes the lower bits and pushes into a 128 bit long long

2d (8 minutes):

```
f4:  movl    $1, %edx
.L6: rdrand  %rax
     movq    %rax, -8(%rsp)
     cmovc   %edx, %eax
     testl   %eax, %eax
     je      .L6
     vmovsd  -8(%rsp), %xmm0
     ret
```

edx = 1
rax = rand
rax = rsp
cond move if carry flag

```
{
  int result = 0
  while(result == 0) {
    long a;
    int num = _builtin_ia32_rdrand64_step((long long)a);
    result = num;
  }
  long long b = a;
  return
}
```

Sets register to $1 and fills result reg to random bit pattern. Then temporarily moves %rax to rsp and then uses a conditional move if carry flag to set the $ value to the lower bits of resulting register. Then tests if result register to see if need to restart. If test false, sets temp reg to long long 128 bit reg.

2e (15 minutes):

```
f5:   testq   %rdi, %rdi
      jle     .L12
      xorl    %edx, %edx
      vxorpd  %xmm0, %xmm0, %xmm0
.L11: rdrand  %rax
      vmovq   %rax, %xmm1
      rdrand  %rax
      vmovq   %rax, %xmm3
      rdrand  %rax
      vdivsd  %xmm3, %xmm1, %xmm1
      vmovq   %rax, %xmm2
      addq    $1, %rdx
      vaddsd  %xmm2, %xmm1, %xmm1
      vaddsd  %xmm1, %xmm0, %xmm0
      cmpq    %rdx, %rdi
      jne     .L11
      ret
.L12: vxorpd  %xmm0, %xmm0, %xmm0
      ret
```

Uses 3 registers to get 3 different random #s. Then it divides the 2nd and 1st rand and stores into 1st var. Then it adds 1 to a counter and adds the 3rd rand with the new first one, store into 1st. Then puts it into a reg of xmm0.
If counter != rdi, redo loop.
If is, end.

```
f5 (int counter) { if (counter <= 0)
int i;
while(i != counter) return;
    long a, b, c;
    int num = __builtin__ia32_rdrand64_step ((long long)a);
        num = __builtin__ia32_rdrand64_step ((long long b);
        num = __builtin__ia32_rdrand64_step ((long long)c);

    long long a1, b1, c1;
    a1 = a; b1 = b; c1 = c;
    a1 = b1/a1;
    a1 = c1 + a1;
    long long result = a1;
      i++;
}
return;
}
```

3 (6 minutes). If function f4 is considered to be a function that returns random numbers, how are these numbers distributed? That is, what is the relative likelihood of it returning one number (say 0.5) versus some other number (say, 65536.5)? How about the relative likelihood of it returning a value in the range [0, 1) versus [65536, 65537)? Approximate answers are OK here. (The notation [a, b) stands for numbers x such that a≤x<b.) (If you're having trouble analyzing f4, analyze f3 instead for part credit.)

The way the number grabs random numbers is by putting in random number bits and then checking. Therefore, the relative likelihood is around the same for every number, except for 0. This is because 0 will occur everytime rdrand fails, which is a higher likelihood of choosing any specific #. Therefor the relative likelihood of [0,1) is more likely than [65536 65537) because it failing.

4 (4 minutes). The function f4 accesses storage that is past the top of the stack. Briefly explain why this isn't dangerous in this context.

It's not dangerous because in the earlier movq instr, we allocate space -8 past the stack, so accessing -8 later is initialized.

5 (6 minutes). The function f4 is inefficient. Optimize it to use as few x86-64 instructions as possible, without changing what it does.

6 Assume the function f5 is being used in a compute-intensive way, and is the bottleneck in your computation. You would like to improve its performance. When you tune it, it is OK if the faster version returns a different answer from the original version; you won't quibble about rounding errors or entropy exhaustion going differently. But you will want the same number of random numbers (or zeros, for rdrand failures) to be generated.

6a (16 minutes). Rewrite f5's assembly-language implementation so that it will run faster by exploiting instruction-level parallelism. Briefly explain the method you're using to transform the code to make it faster, and why it should work. Give a very rough estimate of how much faster you think it'll go and why. List any assumptions you're making

When looking at f5's implementation. The bottle neck of the code is the division step because it is the most complex. Therefore, we should run all of our other instructions while the division is happening to make it faster.

Also, instead of implementing and moving a variable one at a time, maybe CPUs can do multiple moves at same time so we can clump the moves together and set the generated randoms to separate registers

```
f5: testq %rdi, %rdi
    vxorpd %xmm0, %xmm0, %xmm0   ← only called once to set to 0
    jle .L2
    xorl %edx, %edx
.L11 rdrand %rax
    rdrand %rdi   ← arbitrary reg.   ) group rdrand and vmovq
    vmovq %rax, %xmm1
    vmovq %rdi, %xmm3
    vdivsd %xmm3, %xmm1, %xmm1
    rdrand %rax
    vmovq %rax, %xmm2                ) ← do other steps while vdivsd runs
    addq $1, %rdx
    vaddsd %xmm2, %xmm1, %xmm1       ← switch to vmovq because adding to 0 from above is same as moving
    vmovq %xmm1, %xmm0
    cmpq %rdx, %rdi
    jne .L11
    rep
.L12 ret
```

This should go a bit faster because parallelism and less instr and heavy functions

6b (8 minutes). Can you use the x86-64 SIMD to make f5 go faster instead? If so, briefly explain a very rough estimate of how much speedup you'd expect and why. If not, explain why not. Again, list any assumptions.

Yes, you can use SIMD on instructions like the division of the two 128 bit numbers, speeding up the time of that result. Since that is the biggest bottle neck, also with creating the random numbers, the speed up would be pretty good, depending on how many processes help compute the division.

This assumes the division is the biggest bottleneck by far.

6c (10 minutes). Can you use MIMD to make f5 go faster instead? If so, should you use multithreaded, multiprocessed, or multiplexed MIMD? For your chosen method, briefly explain why you chose it over the others, very roughly how much speedup you'd expect, and whether and why your speedup will exhibit strong scaling or weak scaling. List any assumptions.

I would choose multiprocessed because we are doing large computations with large bit registers, meaning having multiple processes would allow to have their own stored registers and RAM, while avoiding race conditions because they don't share as much competing data.

This is better than multithreading because multithreading runs into a lot of race conditions, and even though the communication between threads is faster, for this problem we need to make sure each value is set well.

A multiplexed MIMD wouldn't help as much because it still uses one CPU and the bottle neck is still very prevalent.

This can exhibit strong scaling because it takes more machines to do more insns.