Austin Herring

CS 540 Homework 2

October 8, 2015

1. The tux machines use the AMD Opteron Processor 6136. They have 65972712 kB of total memory available.

2. The processor is intended as a server processor. They have a frequency of 2,400 MHz and a bus speed of 3,200 MHz, and they use a microarchitecture of “K10.” The data width is 64 bits, making it a 64-bit processor, and it contains 8 cores, supporting 8 threads in total. There are two sets of 8 L1 caches, one each for instructions and data, at 64 KB apiece. Additionally, there are 8 512 KB L2 caches and 2 6 MB L3 caches. Finally, in physical memory, the processors support 128 GB, per socket.

3. Using tux as a reference machine, compiling loop.c using gcc with the -S flag produces the following code for the function:

```
loop:
.LFB0:
   .cfi_startproc
   pushq %rbp
   .cfi_def_cfa_offset 16
   .cfi_offset 6, -16
   movq  %rsp, %rbp
   .cfi_def_cfa_register 6
   movq  %rdi, -24(%rbp)
   movq  %rsi, -32(%rbp)
   movq  %rdx, -40(%rbp)
   movl  %ecx, -44(%rbp)
   movl  $0, -4(%rbp)
   jmp   .L2
.L3:
   movl  -4(%rbp), %eax
   cltq
   leaq  0(,%rax,8), %rdx
   movq  -24(%rbp), %rax
   addq  %rdx, %rax
   movl  -4(%rbp), %edx
   movslq   %edx, %rdx
   leaq  0(,%rdx,8), %rcx
   movq  -32(%rbp), %rdx
   addq  %rcx, %rdx
   movsd (%rdx), %xmm1
   movl  -4(%rbp), %edx
   movslq   %edx, %rdx
   leaq  0(,%rdx,8), %rcx
   movq  -40(%rbp), %rdx
   addq  %rcx, %rdx
   movsd (%rdx), %xmm0
   addsd %xmm1, %xmm0
   movsd %xmm0, (%rax)
   addl  $1, -4(%rbp)
.L2:
   movl  -4(%rbp), %eax
   cmpl  -44(%rbp), %eax
   jl .L3
   popq  %rbp
   .cfi_def_cfa 7, 8
   ret
   .cfi_endproc
```

First, before the analysis, note that the loop in the original C code must execute n times; this is because the loop condition is “`i < n`,” and i starts at 0 and increases by 1 every loop iteration.

Therefore, calculating sum sum from 0 to n – 1 of 1 gives (n – 1 – 0 + 1) = n, the number of loop iterations.

Now, for the assembly instructions, ignoring all directives, which start with ".", the instructions "`pushq %rbp`" through "`jmp .L2`" are 8 instructions that will be executed only once. Then, jumping to `.L2`, the first three instructions are equivalent to the "`i < n`" condition of the for loop in the C code, meaning that they will be executed once for every iteration of the loop plus one last time, when the test fails and the loop exits. Because the loop executes n times, these instructions account for 3 x n + 3 = 3 x (n + 1) instructions in total. Next, the instructions between `.L3` and `.L2` are the loop body, and as justified above, the loop will execute n times. There are 20 instructions in the loop body, meaning these will account for 20 x n total instructions. Finally, after the "`jl .L3`" instruction after the `.L2` label, there are two remaining instructions, "`popq %rbp`" and "`ret`." Summing these all gives a total of:

8 + 3(n + 1) + 20n + 2 = 10 + 3n + 3 + 20n = 23n + 13 instructions

4.

<u>countloop</u>

Unoptimized:

Sample runs:

```
Enter vector size: 0
Number of instructions = 256
Number of fp operations = 0
Number of instructions = 256
Number of fp operations = 0

Enter vector size: 1
Number of instructions = 280
Number of fp operations = 1
Number of instructions = 280
Number of fp operations = 1

Enter vector size: 10
Number of instructions = 487
Number of fp operations = 10
Number of instructions = 487
Number of fp operations = 10

Enter vector size: 100
Number of instructions = 2557
Number of fp operations = 100
Number of instructions = 2557
Number of fp operations = 100

Enter vector size: 200
Number of instructions = 4857
Number of fp operations = 200
Number of instructions = 4857
Number of fp operations = 200

Enter vector size: 400
Number of instructions = 9457
Number of fp operations = 400
Number of instructions = 9457
Number of fp operations = 400

Enter vector size: 800
Number of instructions = 18656
Number of fp operations = 800
Number of instructions = 18657
Number of fp operations = 800

Enter vector size: 1000
Number of instructions = 23256
Number of fp operations = 1000
Number of instructions = 23257
Number of fp operations = 1000
```

As can be seen here, for 0 elements, the “baseline” number of instructions is 256. Then, for 1, the number of instructions is 280. Subtracting the “baseline” 256 from that gives 24 instructions, which is approximately 23 x 1 + 13 instructions calculated in question 3, particularly if the constant factor of 13 is dropped (because this 13 is subsumed by the 256 subtracted as a “baseline”). Additionally, from here, the number of instructions can be seen to increase linearly: multiplying the vector size by 10 does approximately the same to the number of instructions, and the same holds true for multiplying the vector size by 2. Note that the number of floating point instructions always equals the vector size, because for all input sizes, the sum `c[i] = a[i] + b[i]` must be calculated for each `i`, and each is one floating point operation.

Optimized:

Sample runs:

```
Enter vector size: 0
Number of instructions = 242
Number of fp operations = 0
Number of instructions = 242
Number of fp operations = 0

Enter vector size: 1
Number of instructions = 250
Number of fp operations = 1
Number of instructions = 250
Number of fp operations = 1

Enter vector size: 10
Number of instructions = 304
Number of fp operations = 10
Number of instructions = 304
Number of fp operations = 10

Enter vector size: 100
Number of instructions = 844
Number of fp operations = 100
Number of instructions = 844
Number of fp operations = 100

Enter vector size: 200
Number of instructions = 1444
Number of fp operations = 200
Number of instructions = 1444
Number of fp operations = 200

Enter vector size: 400
Number of instructions = 2644
Number of fp operations = 400
Number of instructions = 2644
Number of fp operations = 400

Enter vector size: 800
Number of instructions = 5043
Number of fp operations = 800
Number of instructions = 5044
Number of fp operations = 800

Enter vector size: 1000
Number of instructions = 6243
Number of fp operations = 1000
Number of instructions = 6244
Number of fp operations = 1000
```

Here, with the zero vector size, it can be seen that the “baseline” number of instructions is 242 is already lower than the 256 for the unoptimized version. For all bigger vector sizes, the number of instructions remains strictly less than the corresponding optimized version as well. However, the mostly-linear relationship between vector sizes remains; that is, increasing the vector size by a factor of 10 does the same to the number of instructions, approximately, and the same holds true for doubling the vector size and the number of instructions. Additionally, the number of floating point operations remains the same, because the all of the sums of the vectors must still be calculated.

<u>timeloop</u>

Unoptimized:

Sample runs:

```
Enter vector size: 0
Number of cycles = 324

Enter vector size: 1
Number of cycles = 378

Enter vector size: 10
Number of cycles = 1130

Enter vector size: 100
Number of cycles = 7380

Enter vector size: 200
Number of cycles = 14498

Enter vector size: 400
Number of cycles = 29369

Enter vector size:  800
Number of cycles = 58299

Enter vector size: 1000
Number of cycles = 72774
```

For the unoptimized timeloop program, it can be seen that the baseline number of cycles the processor must take to "compute the sums" (of which there actually aren't any) is 324. Next, for a vector of size 1, the number of cycles is 378; subtracting out the baseline 324, it can be seen that the number of cycles should grow by about 378 – 324 = 54 cycles per increase of 1 in n. For a size of 10, the number of clock cycles is approximately 10 times that of 1, which is expected for the sort of linear relationship extracted in part 3. In fact, subtracting out the baseline 324 and dividing by the approximate number of cycles per vector element retrieved from the vector of size 1, 54, gives about (1130 – 324) / 54 = 14.926 = ~15. While not exact, this is a fairly close approximation of the vector size of 10; the discrepancy can perhaps be explained by things like stalls that increase in frequency as the vector size increases. In any case, as the vector size continues to increase, the number of cycles continues to grow linearly, with an increase in vector size by a factor of 2 or 10 having the approximately same effect on the number of cycles.

Optimized:

Sample runs:

```
Enter vector size: 0
Number of cycles = 303

Enter vector size: 1
Number of cycles = 319

Enter vector size: 10
Number of cycles = 658

Enter vector size: 100
Number of cycles = 1289

Enter vector size: 200
Number of cycles = 2333

Enter vector size: 400
Number of cycles = 3955

Enter vector size:  800
Number of cycles = 7682

Enter vector size: 1000
Number of cycles = 9704
```

For the optimized timeloop program, the baseline for a zero-size vector is 303. For a vector of size 1, the number of cycles is 319, so subtracting the baseline out, as before, gives an approximate per-element count of 319 – 303 = 16. Both the baseline and, especially, the per-element count already show a significant decrease in comparison to the unoptimized program. As above, moving on to size 10, the number of cycles is approximately an order of magnitude higher, though slightly less than that, and, using the numbers as calculated above, calculates out to what would be about (658 – 303) / 16 = 22 elements, which is a bit of an overestimate. Doing the same for 100 elements, however, gives only 66 elements. However, despite these aberrations, which can again be accounted for as above for the unoptimized version, the number of cycles still increases, very approximately, linearly, with increases by a factor of 2 or 10 in the vector size being mirrored by the number of cycles taken.