CS194-15 Assignment 1: Measuring CPU Performance

Created by: Patrick Li, David Sheffield

Due: September 14, 2015

Logistics

- Hand in a report containing your answers to all of the questions outlined in the *Task* sections. We expect roughly 1-2 sentences for 2 point questions. 2-3 sentences for 3 point questions. 3-4 sentences for 4 point questions. Questions requiring coding will be worth more points.
- There is a 10% grade penalty for each day late, up to a maximum of 7 days, after which we will no longer accept your assignment.
- Each student is allowed 5 days slack for the year which may be used for postponing handing in an assignment with no penalties. However, a maximum of 7 days late still applies to every assignment.
- At the end of the semester, any slack days unused can be redeemed for a bonus 10% grade on any assignment.

1 Measuring Execution Time

In this part we will be writing a simple C program to measure the amount of time required to compute the sum of the integers from 0 to 10000. The Linux performance tools library allows us to accurately count the number of cycles it takes to execute a chunk of code.

Two files are given to you, counters.h and counters.cpp, which take care of setting up and reading the hardware performance counters.

1.1 C Skeleton

The following code snippet shows how to use the provided counter library. This code shows how to initalize a hardware counter, and also how to retrieve the current time in cycles.

```
#include <cstdlib>
#include <cstdio>
#include <cstring>
#include <time.h>
#include "counters.h"

int main(int argc, char *argv[])
{
  //Initialize a hardware counter
  hwCounter_t c1;
  c1.init = false;
  initTicks(c1);

  //Get current time in cycles
  uint64_t current_time = getTicks(c1);
  printf("Current Time in Ticks %lu\n", current_time);
}
```

Compile the above snippet by saving it to skeleton.cpp and typing the following command in the shell.

```
g++ -msse4 -O2 counters.cpp skeleton.cpp -o skeleton
```

The -msse4 flag tells the compiler to use the SSE instructions when possible. The -O2 flag tells the compiler to run most standard optimizations. The -o option tells the compiler to generate a binary executable named "skeleton". After compilation, run the executable by typing the following.

```
./skeleton
```

This should print out something similar to the following.

```
Current Time in Ticks 9300
```

Take note that the time returned by getTicks() is meaningless by itself. In order to measure the time *elapsed* during the execution of some code, we must call getTicks() twice. Once before execution, and once after, and then compute the difference between the two times.

```
//Get time before
uint64_t time = getTicks(c1);

... do some computation ...

//Get time after
uint64_t elapsed = getTicks(c1) - time;
```

Use this provided skeleton to time the number of cycles required to compute the sum of the integers from 0 to 10000 (exclusive).

1.2 Computing the Sum

Use the following code snippet to compute the sum.

```
int main(int argc, char *argv[])
{
  //Get the number of the user
  int n = atoi(argv[1]);

  //Compute sum
  long long sum = 0;
  for(long long i=0; i<n; i++)
    sum += i;
}
```

Save this snippet to a file called sum.cpp, and compile it using:

```
g++ -msse4 -O2 sum.cpp -o sum
```

And run it using:

```
./sum 10000
```

1.3 Tasks

Use what you learned from section 1.1 and 1.2 to time the execution for computing the sum from 0 to 10000 (exclusive). *Remember to print out the result* of the sum *after* computing the elapsed time. Answer the following questions:

- With -O2 enabled, how many cycles does it take to compute the sum?
- Without -O2 enabled, how many cycles does it take to compute the sum?

Now try repeating the above experiments but *without* printing out the results of the sum.

- With -O2 enabled, how many cycles does it take to compute the sum if you don't print the result?
- Without -O2 enabled, how many cycles does it take to compute the sum if you don't print the result?

In section 1.2 you might notice that we read in the number 10000 as an input from the user. What happens if we instead hardcode the number directly into the program like so:

```
//Compute sum
long long sum = 0;
for(long long i=0; i<10000; i++)
  sum += i;
```

Now how long does it take to execute the sum?

To summarize, there are three axis to explore, whether -O2 optimization is on or off, whether you print out the results of the sum or not, and whether you accept the number 10000 from the user or whether it is hardcoded. Time all 8 configurations of the above options (3×8 points) and explain the discrepancy in the timings (2 points). Which configuration is the most accurate way of timing the computation for the sum (2 points).

Also how does "atoi(argv[1])" work? What is "argv[1]" (1 point) and what does "atoi()" (1 point) do?

2 Measuring Memory Latency

In this part we will be writing a benchmark for measuring the number of cycles between requesting and obtaining a value from memory, also known as the memory latency. We will use a technique called "pointer chasing" to do this.

2.1 Random Values

Before we explain the pointer chasing benchmark, we need to be able to create a shuffled array. In order to do this, we need to be able to generate random integers first. The following code snippet shows how to initialize the random number generator and generate a random integer between 0 and 10.

```
#include <cstdlib>
#include <cstdio>
#include <cstring>
#include <time.h>
#include "counters.h"

int main(int argc, char *argv[])
{
  //Seed the random number generator with the current time
  srand(time(NULL));

  //Generate a random integer between 0 and 10
  int n = rand() % 10;
}
```

Use this code snippet to generate an integer array of length, N, with the integers $\{0 \ldots N-1\}$ in a randomly shuffled order.

2.2 Pointer Chasing

Assume we start with an array, A, of length N, initialized with the integers $\{0 \ldots N-1\}$ in shuffled order.

- Initialize the starting value of our index, i, to 0.

- In the first step, update i by setting it to the number contained in $A[0]$. For this example, let us say that i is now 3.
- In the second step, update i by setting it to the number contained in $A[3]$. Let us say that i is now 7.
- In the third step, update i by setting it to the number contained in $A[7]$.
- Continue in this fashion for the required number of steps.

Program a pointer chasing benchmark that follows the steps outlined above. Allow the length, N, of the array to be inputted by the user. Time how long it takes to run the simulation for 2^{20} steps, and compute the *average* number of cycles to complete a single step. Remember to keep in mind the subtleties involved in timing a computation you learned from part 1.

2.3 Tasks

- Explain how this pointer chasing benchmark measures the time between requesting and obtaining a value from memory (i.e. the memory latency). (2 points)
- How many bytes in memory is an int array of length N on the computer you are running on? (1 point)
- Generate a plot showing the relationship between the average number of cycles to execute a step (vertical axis) and the size in bytes of the array (horizontal axis). Vary the size of the array from 32 kilobytes to 8 megabytes. (32 points)
- Discuss the shape of the curve. In particular, explain what the sudden slope changes in the curve correspond to. (4 points)
- In class we discussed how modern out-of-order processors can execute many instructions in parallel through the use of pipelining. Thus even if a single instruction takes t nanoseconds to complete, ten instructions will likely take much less than $10t$ nanoseconds to complete. Does this affect our benchmark? That is, if it takes t cycles to run c steps in our pointer chasing benchmark, how can we be sure that a single step is actually completing in $\frac{t}{c}$ cycles and that the steps aren't simply being pipelined? (2 points)
- BONUS: We would ideally like the index, i, to eventually traverse through the entire array. That is, we would like i to take on all the values between 0 and N eventually. However not every array A allows this. Write a function that when given an array A, computes how many unique values i actually takes on. Explain how the code works. (16 points)

3 Measuring Memory Bandwidth

In this part we will be writing a benchmark for measuring the average number of bytes that the memory system can transfer to the CPU per second. This is known as the memory bandwidth. The two most important parameters that characterize a memory system is its latency and its bandwidth.

3.1 Measuring Time in Seconds

In order to measure time in seconds, instead of in cycles as we did previously, we need a new timing routine. The following code prints out the current time measured in seconds.

```
#include <sys/time.h>
#include <time.h>
#include <cstdio>

int main(int argc, char *argv[])
{
  struct timeval tv;
  gettimeofday(&tv, 0);
  double time = tv.tv_sec + 1e-6 * tv.tv_usec;
  printf("Current time = %f seconds.\n", time);
}
```

3.2 Array Copying

In order to measure the memory bandwidth of your computer system, we are going to time how long it takes to copy an array. Write a program that given an array of length, N, calculates the average number of megabits copied per second (Mbps). Remember to first "warm up the cache" by copying the array a few times before starting timing.

3.3 Efficient Array Copying

We have provided two different efficient array copying procedures in "simd_copy.cpp", simd_memcpy and simd_memcpy_cache. These two routines use SSE intrinsics, to maximize the performance. Alter your program to be able to use any of the three array copying procedures to time how long it takes to copy an array of length, N.

3.4 Tasks

- Plot the relationship between the average bandwidth (in Mbps) sustained by the memory system and the total size (in bytes) of the array. Produce a plot using each of the three copy routines, your own routine,

simd_memcpy and simd_memcpy_cache. Vary the size of the array from 32 kilobytes to 8 megabytes. (3×8 points)

- Discuss the shape of the curve. Explain any sudden increases in bandwidth as the array size varies. (4 points)
- Explain why it is necessary to first "warm up the cache" by copying the array a few times before starting timing. (2 points)
- We are trying to measure the *maximum* bandwidth that the memory system can sustain. Explain why an inefficient array copying procedure would yield an inaccurate measure of the bandwidth. (2 points)
- The simd copying procedures use SSE intrinsics to improve the performance. Intel and AMD processors come with a special set of instructions, called SSE instructions, for directly controlling the simd functional units. SSE intrinsics are a pleasant C wrapper for these instructions, so that the user does not need to write assembly code to gain access to this functionality. Find all the different SSE intrinsics calls (they start with "_mm_") and explain what each of them do. The "Intel Intrinsics Guide" is a good resource. (2×3 points)

4 Measuring Flops and IPC

In this part we will be writing a benchmark for measuring the number of floating point operations completed per second (Flops) of matrix multiply and the number of instructions completed per cycle (IPC). Flops is an important performance measure for numerical code.

4.1 Measuring Instructions

The provided counter library can measure the number of instructions executed in a chunk of code in a similar manner to how we measured the number of cycles elapsed.

```
//Initialize instruction counter
hwCounter_t c;
c.init = false;
initInsns(c);

//Get current instruction count
uint64_t count = getInsns(c);

.... do some computation ....

//Compute number of instructions executed
uint64_t executed = getInsns(c) - count;
```

We will be using this feature to measure the IPC of four different implementations of matrix multiply.

4.2 Matrix Multiply

We are providing a binary file, square_sgemm.o, that contains four different implementations of square matrix multiply (opt_simd_sgemm, opt_scalar1_sgemm, opt_scalar0_sgemm, and naive_sgemm), and a skeleton file, mmultiply.cpp, that shows how to call them.

Here is the prototype for naive_sgemm:

```
void naive_sgemm(float *Y, float *A, float *B, int n);
```

Naive_sgemm takes two square matrices, A and B, and their width, n, as input and stores the result of multiplying A and B in Y. The other implementations have identical signatures.

Write a program that computes the average number of floating point operations completed per second (Flops) and the number of instructions completed per cycle (IPC) for multiplying matrices of size 2^{10}.

4.3 Tasks

- Produce a table recording the Flops and IPC of each matrix multiply implementation. (6×4 points)
- How many floating point operations (in this case, multiply and add) are performed to multiply two square matrices of size n? (3 points)
- Compute the Flops by dividing the total number of floating point operations performed by the time in seconds required.
- Some of the implementations should have measured IPC exceeding 1, indicating that the processor is completing *more* than one instruction per cycle. How can this be? (2 points)
- Comparing opt_scalar1_sgemm with opt_simd_sgemm, opt_simd_sgemm should have a *lower* measured IPC than opt_scalar1_sgemm even though it performs nearly eight times more floating point operations per second. How can this be? Isn't a higher IPC always an indicator of an efficient program? (2 points)