Yahya Alhinai

EE5371

September 23, 2019

Problem 1:

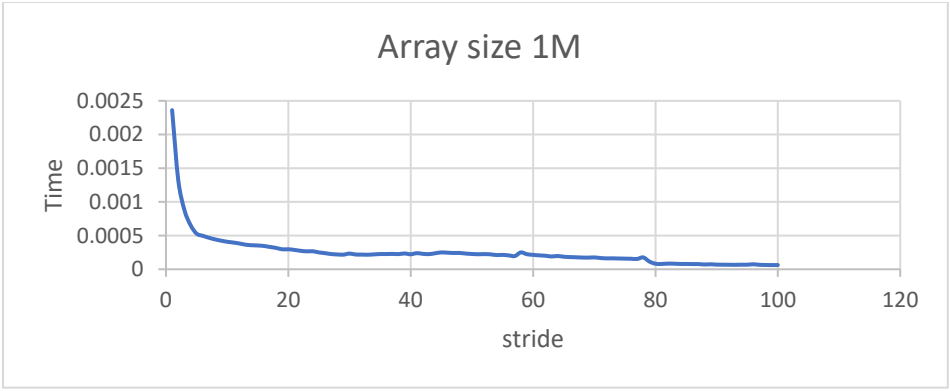
- After reading "*Considerations When Evaluating Microprocessor Platforms*", I tend to lean towards the application point of view rather than the hardware point of view. As stated in the paper, application developers have the privilege to redesign and change their application to be optimal at doing the things that supposed to do. The most important advantage that was mentioned in the paper was that developer can evaluate specific hardware and maximize hardware's potential to provide advancement in a particular domain of need. This is the reason why application developers reported up to 100x speed up from using GPU over CPU. A program might has been optimally optimized to run on GPUs and fully utilize their potential. The same program might have been performed on CPU to compare the speedup, leaving the fact it has completely different architecture that is designed to deal with different functionality. As it was mentioned in the paper, "application developers could have also achieved large performance improvements by rewriting their legacy code to target parallel CPU implementations". Meaning the gap difference in performance that was reported could have vanish has developers optimized two versions of the program aimed for two different hardware architecture.

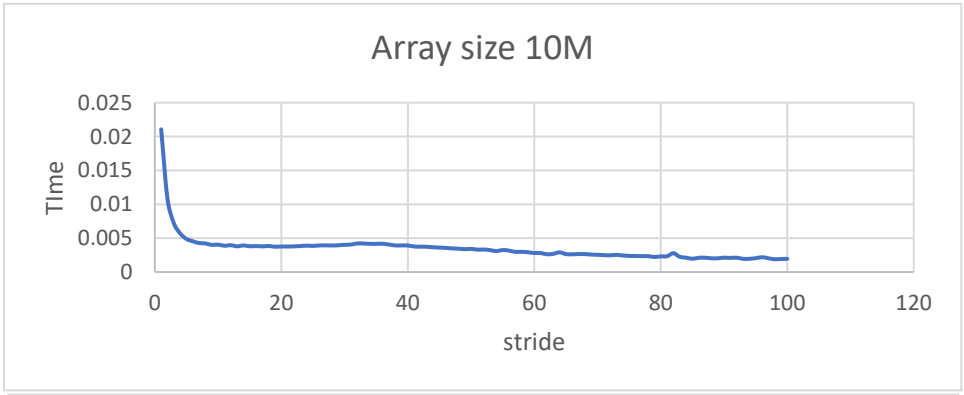
Problem 2:

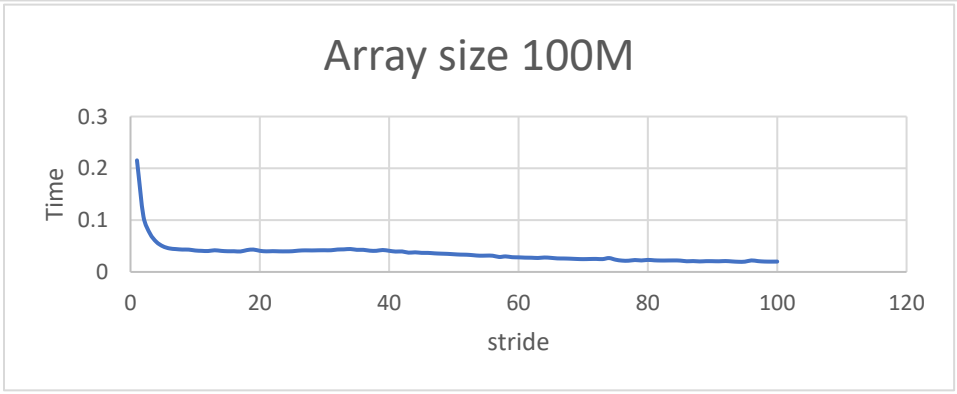
- It was interning reading the paper "*How not to lie with statistics: the correct way to summarize benchmark results*" by Fleming and Wallace [2]. The paper shows in rule 1 the drastic difference in the arithmetic mean values as the normalized reference changes and the superiority of geometric mean in averaging normalized numbers: rule 2. The authors of this paper never mentioned the necessity of using normalized data over working with the raw data. That's why I find myself leaning towards the second paper "*Characterizing Computer Performance with a Single Number*" by Smith [3]. This paper, as well as the two textbooks of this course, has emphasized the different usage of arithmetic, harmonic and geometric mean depends on the data itself. For instance, arithmetic mean is an accurate measure of performance expressed as time because performance is directly proportion to the sum of inverses of the times. On the other hand, performance in arithmetic mean is not inversely proportion to the sum of times; therefore it should not be used to represents performance expressed as a rate. Rather, harmonic mean should be used due to the face that the performance in harmonic mean is inversely proportion to the sum of times, but not to measure of performance expressed as time. Lastly, the performance in the geometric mean does not have an inverse relationship with the total time and as a result it shouldn't be used to summarize the data. One area geometric mean is useful is in normalized data because it maintains consistency as the normalized reference changes.

Problem 3:

1.





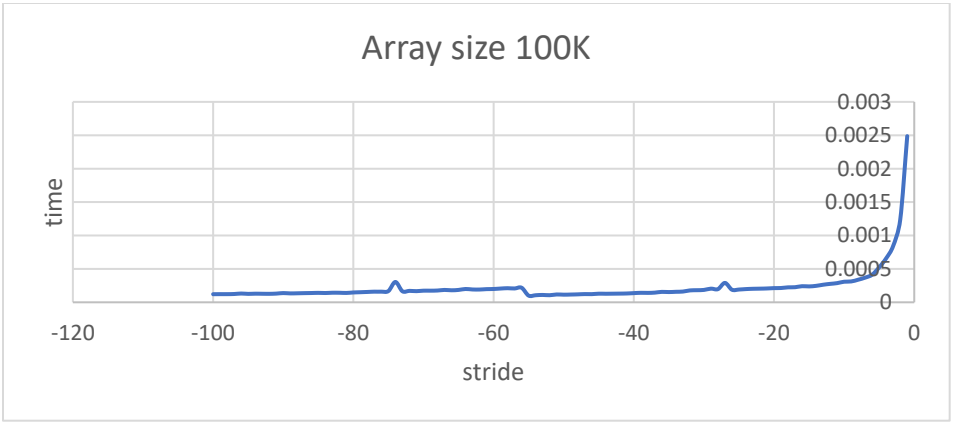


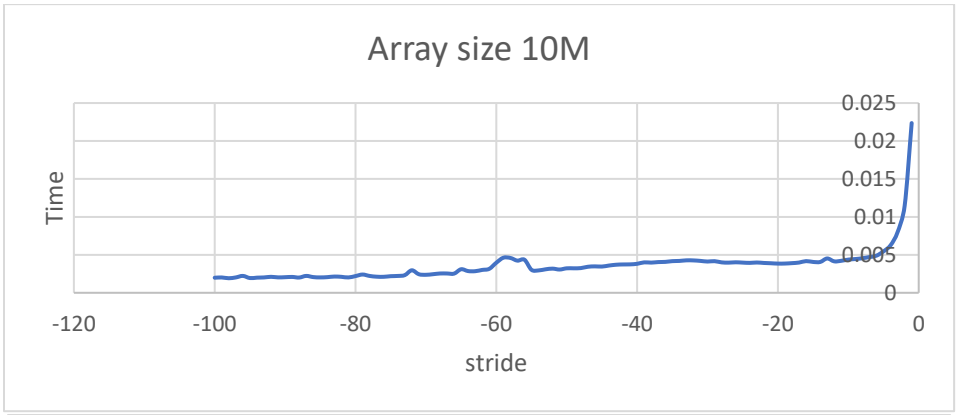
- Theoretically, the relationship between the number of stride vs the time it takes supposed to be linear. Described by the following equation:

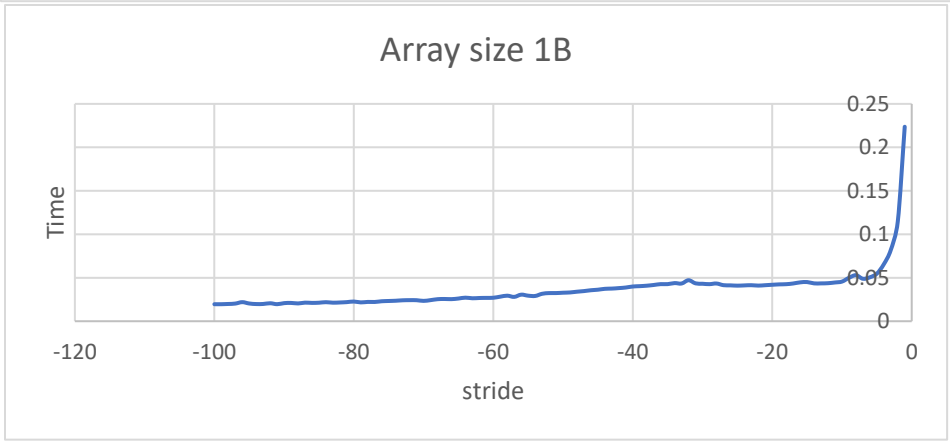
$$Time = \frac{time\ it\ takes\ to\ scan\ ever\ elemnt\ in\ the\ array}{stride}$$

- The obtained data represented in the graphs above. It deviates from the linear behavior as the as the number of strides increases. This is because as the number increases, arithmetic operations involving multiplication and addition needed to land on the desired index. As well as, it's desirable cash-wise to access adjacent indexes rather that spread apart indexes.

2.







- As the stride has a negative value, the function stride vs time behaves the same as if stride has a positive value. Thought, it has been observed that the total time to go through all the array with descending stride has 3-6% increased in comparison to part A. The is an indication that multiplication and then subtraction is more expensive instruction-wise in comparison to part A.

3. The following snip of code was as a benchmark program to measures the maximum MIPS value of the machine it is running on:

```
long unsigned int n = 10000000000;
long unsigned int * x = new long unsigned int[n];

for (int i = 0; i < n; i++){
    x[i] = 1;
}
```

To measure the maximum MIPS value, the following set of MIPS instructions is describing the behavior of the above C++ code and the loop runs 10 billion time:

```
loop:
    sltu   $t3, $t0, $t1        # i < n
    beq    $t3, $zero, done     # done?
    sw     $t2, 4($t0)          # A[i] = 1
    addi   $t0, $t0, 1          # i++
    j      loop                 # loop n times
done:
```

several assumptions were made:

- The C++ code was optimally converted to MIPS instructions with the least amount of instructions.
- No pipeline collisions.
- The whole program is running on a single core.
- The program is utilizing the max frequency at 4.10GHz to run the program

ideal running time:

$$Total\ Time = \frac{5\ MIPS\ instr * 10^{10}\ times}{4.10 * 10^{9}\ Hz} = 12.195122\ Seconds$$

Tested running time:

$$Total\ Time = 12.399628\ seconds$$

The test running time is expected to be more than or equal to the ideal running time. The real running time was 98.35% of the ideal running time. There are several reasons for this. One of them is the operating systems uses some of the computation power. Also, the max speed frequency of a single code is not fixed at 4.10GHz; the max speed frequency was observed to be fluctuating between 3.98 to 4.08GHz. Lastly, since there is a great deal if element, acceding data from memory might take more than expected.

4. benchmark program to measures the maximum MFLOPS value:

```
7.  long unsigned int n = 100000000;
8.  double * x = new double [n];
9.
10. for (int i = 0; i < n; i++){
11.     x[i] = x[i]*x[i];
12. }
```

The following set of MFLOPS instructions is describing the behavior of the above C++ code and it loops around 100 million times:

```
8.  loop:
9.      sltu    $t3, $t0, $t1        # i < n
10.     beq     $t3, $zero, done     # done?
11.     lw      $t4, 4($t0)          # t4 = A[i]
12.     mul     $t5, $t4,$t4         # t5 = t4 * t4
13.     sll     $t6
14.     sll     $t7
15.     Addu    $t5, $t6, $t7
16.     sw      $t5, 4($t0)          # A[i] = t5
17.     addi    $t0, $t0, 1          # i++
18.     j       loop                 # loop n times
19. done:
```

The same assumptions of Part 3 are present here.

ideal running time:

$$Total\ Time = \frac{10\ MIPS\ instr * 10^8\ times}{4.10 * 10^9\ Hz} = 0.243902\ Seconds$$

Tested running time:

$$Total\ Time = 0.261527\ seconds$$

The real running time was 93.26% of the ideal running time. Besides the computation power operating system uses and the clock speed fluctuating, the main restriction for not getting an ideal running time is accessing memory. The reason loop runs only 10^8 times instead of 10^{10} times as part 3 is because it required a great deal of memory. The imperfection in performance is highly linked with cash misses and collisions and acceding RAM.