Homework 2: Hardware and Code Optimization: Calculating Pi

Justin Campbell
UT eID: jsc4348
SDS 335
Dr. Eijkhout

November 23, 2021

1. Instructions: Setup

The full path to the directory where the experiment was conducted is

$$/home1/07674/jcam98/hwk\_2\_code\_op$$

2. Calculating Pi

Problem Statement: As discussed in class, the following python instructions can be used to calculate pi

```
[In [1]: import numpy as np
[In [2]: points = np.random.random((10000,2))
[In [3]: points2 = np.square(points)
[In [4]: norm2 = np.sum(points2, axis=1
[In [5]: num_inside = np.sum(norm2 <= 1.0)
[In [6]: 4.0 * num_inside / 10000.0
[Out[6]: 3.1256
```

a) Problem Statement: Explain how the algorithm works.

Solution:
In the first line of the algorithm, the numpy module, (which contains a wide range of mathematical/arithmetic operations for matrices in python), is imported into the workspace. Then, a 10000 by 2 matrix is initialized with random floating point values between 0 and 1. Each of the elements in the matrix are squared and the elements along the 1st axis are summed to obtain the norm2 of the matrix. Then, each of the elements in the norm2 vector are summed if their value is less than or equal to one. This result is multiplied by 4 and divided by the number of rows of the ”points” matrix, ”10000.0” to obtain an approximation for pi.

b) Problem Statement: What size and shape are the arrays "points", "points2", and "norm2"?

Solution:

- points: 10000 row by 2 column matrix
- points2: 10000 row by 2 column matrix
- norm2: 10000 row by 1 column vector

3. Programming: Part 1

Problem Statement: Write code in your favorite language, i.e., C/C++ or Fortran, that is equivalent to the python code. Specifically, use individual loops for each step of the code. -Use a constant (C/C++) or parameter (Fortran) to set the number of points. For testing, you may use a small number
-If your code does not finish in a reasonable time period for the sample size, then reduce the sample size
- Create the same arrays as in the python code
- Use single or double precision arrays in your code
- Important: Use a "double precision" float or real number for the variable "num˙inside"
- Add a timer to your code. Start the timer after "Step 2", i.e., after the array points has been filled, and stop the timer after "Step 6"

4. Assignment: Part 1

Problem Statement: Add statements to your code so that output is generated that shows what type of precision is being used, how large the sample size was, and how much memory (large arrays only) was used in the arrays (in MB and GB), the value of pi, and how much time the calculation took. Here is an example. The "..." represent the value.

```
Precision=...
Sample size=...
Total memory footprint(MB)=...
Total memory footprint(GB)=...
Pi=3.14...
Timing::...(seconds)
```

Non-Optimized Source Code Solution:

```

//Property of: Justin Campbell; UT eID: jsc4348
//Purpose of Use: SDS 335: Homework 2: Hardware Code and Optimization;
     Calculating Pi                                  \

// Description: This program uses vectors and control structures to develop
     an approximation for the quantity, "Pi" ac\
cording to Part 1 of the assignment with no optimizations. The algorithm
     closely replicates the Python code presented \
in the problem statement.

```

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <vector>
#include <sys/time.h>

using namespace std;

int main (){

  // Define and initialize array elements and loop counters

  //srand (time(NULL));
  double size_double = 10000.0;
  int size_int = 10000;
  struct timeval start, end;
  vector<double>points_x(size_int);
  vector<double>points_y(size_int);

  // Return random value between "0" and "RAND_MAX" and
  // assign to points_x and points_y for each iteration

  int i;
  for (i = 0; i < size_int; i++){

    points_x[i] = double(rand())/double(RAND_MAX);
    points_y[i] = double(rand())/double(RAND_MAX);

  }

  gettimeofday(&start, NULL); // Initialize runtime counter

  vector<double> points_x_2(size_int);
  vector<double> points_y_2(size_int);

  // Increment through "points" vectors and square each element

  for (i = 0; i<size_int; i++){

    points_x_2[i] = pow(points_x[i],2);
    points_y_2[i] = pow(points_y[i],2);

  }

  vector<double> norm2(size_int);

  // Increment through the "points" vectors and add elements in the same row
     to obtain the "norm2" of the matrix

  for (i = 0; i<size_int; i++){

    norm2[i] = points_x_2[i] + points_y_2[i];

  }
  // Increment through "norm2" vector and sum all elements less than or equal
      to 1

  double count;

  for (i = 0; i<size_int; i++){

    if (norm2[i] <=1){

      count +=1;

    }

  }

  // Compute the final approximation for Pi

  double pi_approximation = (4*count)/size_double;

  gettimeofday(&end, NULL); // Terminate run-time counter

  float delta = ((end.tv_sec-start.tv_sec) * 1000000u + end.tv_usec-start.
     tv_usec)/ (1.e6);

  // Compute the memory of large arrays, "points" and "points2" in MBytes

  double large_array_memory = (64.0*size_double*2.0)/(8.0* 1024.0*1024.0);

  // Print the precision of the code, the size of the sample, the memory of
     the large arrays, the approximation for Pi, and the run-time for the
     calculation of the approximation

  cout << "Precision: Double" << endl;
  cout << "Sample Size: 10000" << endl;
  cout << "Total Memory Footprint: "<< large_array_memory << "MB" << endl;
  cout << "Total Memory Footprint: "<< large_array_memory/1024 << "GB" <<
     endl;
  cout << "The approximation of Pi is: "<< pi_approximation<< endl;
  cout << "The elapsed runtime is: "<< delta << endl;

  return 0;
}
```

Terminal Output for Non-Optimized Solution:

```
c203-014[clx](1008)\$ calculating_pi_no_op
Precision: Double
Sample Size: 10000
Total Memory Footprint: 0.152588MB
Total Memory Footprint: 0.000149012GB
The approximation of Pi is: 3.1256
The elapsed runtime is: 0.0001
```

5. Assignment:Part 2

a) Problem Statment: Explain how often data is moved from memory into the CPU (and back!) for the individual steps. Does the cache (or the caches) provide any help to boost performance?

Solution:

The actual and values and memory addresses of the data are moved from memory into the CPU and back for every iteration of a for loop, and each time the program accesses the contents of, and operates on, a vector. The cache boots performance of the code by allowing for values and main memory addresses of data to be temporarily stored in a pipeline during and between iterations for data that needs to be read, operated on, and written back out to main memory from the CPU. This eliminates the need for data to be repetitively sent to and from main memory in successive cycles in the same iteration, thus saving time.

b) Problem Statement: Why would the calculation give an incorrect result for a large sample size (like the one we are using here, see "Part 4"), if the variable "num˙inside" was single precision number?

Solution:

The calculation would give an incorrect result for a large sample size if the variable "num˙inside" was a single precision number because the memory allocated for a single precision number would not be large enough to store the data without round-off errors.

6. Assignment:Part 3

Problem Statement: Copy your code from part 2 into a new file. Perform the following modifications

- Change the part between the calls to the timer
- Do not change how the random numbers are created, nor the arrays "points"
- Change the calculation of "num˙inside", i.e., steps 3 to 5 such that the calculation is done in one loop. This should eliminate using the 2 other arrays, "points2", and "norm2"
- Add output with this information:

```
Pi. = 3.14...
Timing(single loop)::...(seconds)
```

Optimized Solution:

```
//Property of: Justin Campbell; UT eID: jsc4348
//Purpose of Use: SDS 335: Homework 2: Hardware Code and Optimization;
     Calculating Pi
//Description: This program incorporates optimizations to the program in the
     file "calculating_pi_no_op.cc" through
// minimizing the number of loops in the algorithm used to approximate the
     value of "Pi".
```

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <vector>
#include <sys/time.h>

using namespace std;

int main (){

  // Define and initialize array elements and loop counters
  //srand (time(NULL));


                   \
  double size_double = 10000.0;
  int size_int = 10000;
  struct timeval start, end;
  vector<double>points_x(size_int);
  vector<double>points_y(size_int);

  // Return random value between "0" and "RAND_MAX" and assign to points_x
     and points_y for each iteration
  int i;
  for (i = 0; i < size_int; i++){

    points_x[i] = double(rand())/double(RAND_MAX);
    points_y[i] = double(rand())/double(RAND_MAX);

  }

  gettimeofday(&start, NULL); // Initialize runtime counter

  /////////////////////////////  Optimizations are Below
     /////////////////////////
  double x,y,x_2,y_2, sum, count;

  for (i=0; i< size_int; i++){

      x = points_x[i];
      y = points_y[i];
      x_2 = x*x;
      y_2 = y*y;
      sum = x_2 + y_2;

      if (sum<=1){

        count += 1;

      }

  }

  // Compute the final approximation for Pi
  double pi_approximation = (4*count)/size_double;

  gettimeofday(&end, NULL); // Terminate run-time counter

  float delta = ((end.tv_sec-start.tv_sec) * 1000000u + end.tv_usec-start.
     tv_usec)/ (1.e6);

   // Compute the memory of large arrays, "points" and "points2" in MBytes
                                               \

  double large_array_memory = (64.0*size_double*2.0)/(8.0* 1024.0*1024.0);

  // Print the precision of the code, the size of the sample, the memory of
     the large arrays, the approximation for Pi\
, and the run-time for the calculation of the approximation

  cout << "Precision: Double" << endl;
  cout << "Sample Size: 10000" << endl;
  cout << "Total Memory Footprint: "<< large_array_memory << "MB" << endl;
  cout << "Total Memory Footprint: "<< large_array_memory/1024 << "GB" <<
     endl;
  cout << "The approximation of Pi is: "<< pi_approximation<< endl;
  cout << "The elapsed runtime is: "<< delta << endl;

  return 0;

}
```

Terminal Output for Optimized Solution:

```
c203-014[clx](1016)$ calculating_pi_op
Precision: Double
Sample Size: 10000
Total Memory Footprint: 0.152588MB
Total Memory Footprint: 0.000149012GB
The approximation for Pi is: 3.1428
The elapsed runtime is: 7e-06
```

Comments: In comparing the terminal output from the execution of the non-optimized solution with the optimized solution, we can see that the run-time of the pi calculation improved noticeably by a magnitude of about 1.5 * $10^1 from 1.0 * 10^-4 to 7 * 10^-6$.

7. Assignment: Part 4

Problem Statement:
- For your report, set the number of the sample size to 1000000000. Use a smaller number if your codes do not finish in a reasonable time (minutes).
- Run the code on a Frontera Cascade Lake node
- Use the ”numactl -N 0 -m 0 ¡your-executable¿” to run your code

a) Explain what the 2 options and the command do. Use the man page or google

Solution:

The "numactl" command allows for a user to compile and run a program on a specific node (in our case Cascade Lake Node) which is purposeful for "idev". The two flags/options specify the particular node to use.

b) Add the output from "part 1" and "part 3" to your report. Manually add a line that shows how much faster or slower the second code is compared to the first code

Solution: See "Assignment: Part 1" and "Assignment: Part 3" above

c) Include the source code(for both codes) to your report. Do not show the 2 complete codes, but only the parts between the calls to the timer.

Solution: See "Assignment: Part 1" and "Assignment: Part 3" above

8. Assignment: Part 5

a) Problem Statement: Explain/speculate why the execution speed is different between the original and the optimized code

Solution:

The execution speed is quicker for the optimized program because it does not need to repetitively access the same elements in the vectors which is not the case for the non-optimized program. Namely, it merely calls and element from a vector and operates on it a single time, making it easier and quicker to access. Furthermore, the minimization of loops means the actual values and memory addresses of the data used in the computations do not need to be accessed from main memory as many times.