Hardware and Code Optimization: Calculating Pi

Michel Gonzalez

November 23 2021

1 Calculating Pi

In class we discussed about the following code:

```
import numpy as np
points = np.random.random((10000, 2))
points2 = np.square(points)
norm2 = np.sum(points2, axis=1)
num_inside = np.sum(norm2 <= 1.0)
4.0 * num_inside / 10000.0
3.1256
```

This chunk of python based code will plot many **points** on a 2 by 2 square. Then it will count the number of **points** that land within a circle of radius $R = 1$. By taking the ratio of the total amount of **points** and the **points** that are inside the circle we would get the area of that circle which is $A = \pi * R^2 \rightarrow A = \pi$.

The code creates three main arrays named **points**, **points2**, and **norm2**. The first array **points** is a 2D array and has a size of 10000 lists with two elements in each list representing coordinates as (x,y). Each coordinate is between the value of 0 and 1, which means that **points** will be representing the first quadrant of a coordinate system. This is why the last calculation is multiplied by 4 to account for the other quadrants. The second array **points2** is the same as **points** in terms of size however it now holds the squared value of the lists of values in points. This is done to calculate the distance of a point from the origin.

Lastly the array **norm2** will be a 1D array of size 10000 containing the summations of the x^2 and y^2 values in each list element from **points2**. This gives the actual distance from the origin. Once it finishes the summations it then goes

through them and counts all the points that are less than or equal to 1. This is interpreted as counting all the points inside the circle with radius of 1 in terms of our problem. This number is then multiplied by 4 and to account for all quadrants and then divides by the total number of points. This generally produces a decent calculation for π.

2 Analysis

The python code was translated into c++ and each array was made through for loops. This file was complied using **icc –xcore-avx512 -O2 calculating_pi.cpp** and was stored in the following path: **/home1/08376/michel_g/Hw2**. This program was also written in an idev environment in the queue **c210-006[clx]**. The output from my code in c++ is shown below.

```
Percision = d (double)
Sample size = 10000
Total memory foorprint (MB) = 0.4
Total memory foorprint (GB) = 0.0004

Pi = 3.1404
Time measured: 0.000126seconds
```

To analyse how data is being moved from memory and back I will focus on the for loops that build the arrays.

```
for (int i = 0; i <= n; i++) {
   points[i][0] = (double)
   rand()/RAND_MAX;
   points[i][1] = (double)
   rand()/RAND_MAX;
 }
 ////////////////////////////////
 for (int i  = 0; i <= n; i++) {
   points2[i][0] = points[i][0]
   * points[i][0];
   points2[i][1] = points[i][1]
   * points[i][1];
 }
 ////////////////////////////////
 for (int i = 0; i <= n; i++) {
   for (int j = 0; j <= 1; j++) {
     total = total + points2[i][j];
   }
   norm2[i] = total;
   total = 0.0;
```

The code above builds the first array points which in my case has a size of $n = 10000$ elements. The for loop provides two random values between the 0 and 1 and are used as the x and y coordinates. The data is created by the rand() and it is stored into the memory in the array points. Additionally the caches do help boost the overall performance since for this sample size the memory toll is

less than the L2 cache size. This means it can store all the x and y coordinates and to be reused to create the points2 and the norm2 array. Thus, the data is actually mainly coming from the cache but to initialize the data the CPU creates the random value. Additionally these for loops are vectorized since the data is being accessed is independent.

If the num_insde or in my case counter variable was a single precision floating point and we had too large of a sample size this would produce a errors in the computation of the counting. Once the counter has reached its max capacity (for single precision) then it will begin to have round off error and return an incorrect value for num_inside which then gives an incorrect value of pi. At a larger size the value of pi should become more accurate but due to the limitations of memory for floating point numbers it will not work after a certain sample size since there will be too many points that land inside the circle.

3 Alteration

I made a second file with altered code that omitted the points2 and norm2 arrays and did all the calculations and counting within a single loop. I set the sample size to be $n = 1000000000$ and ran the executable with **numactl -N 0 -m 0**. The **numactl** command runs processes with a specific NUMA scheduling or memory placement policy. The **-m** option will only allocate memory from nodes. The **-N** option only executes commands on the CPUs of nodes which can also be be specified.

```
Percision = d (double)
Sample size = 1000000000
Total memory foorprint (MB) = 40000
Total memory foorprint (GB) = 40

Pi = 3.14153
Time measured :: 10.7261 seconds

Percision = d (double)
```

```
Sample size = 1000000000
Total memory foorprint (MB) = 16000
Total memory foorprint (GB) = 16

Pi = 3.14163
Time measured (single loop) :: 1.28031 seconds

Second code is 8.37765 times faster
```

The difference in time it took to execute with a sample size of $n = 1000000000$ between the altered and the original was 9.44579 seconds. This is due to only having to do a single loop of size n instead of three separate iterations to create each array and then an additional for loop to count all the points within the circle. The number of iterations is given by $n_T = n + n + 2n + n$ which amounts to $5n$

iterations. The $2n$ comes from the norm2 array since it has a nested for loop that goes up to $j = 1$. The single loop code is a much better computational method to estimate the value of π.

C++ Code

calculating_pi.cpp

Translation of the python code with all arrays

```
struct timeval start, end;
float delta;
gettimeofday(&start, NULL);

double points2[n][2] = { };
for (int i  = 0; i <= n; i++) {
  points2[i][0] = points[i][0]
  * points[i][0];
  points2[i][1] = points[i][1]
  * points[i][1];
}

double norm2[n];
double total = 0.0;
for (int i = 0; i <= n; i++) {
  for (int j = 0; j <= 1; j++) {
    total = total
    + points2[i][j];
  }
  norm2[i] = total;
  total = 0.0;
}

double counter = 0.0;
for (int i = 0; i <= n; i++) {
  if(norm2[i] <= 1.0) {
    counter = counter + 1.0;
  }
}

counter = 4.0 * counter;
double pi;
pi = counter / (double) n;

gettimeofday(&end, NULL);
delta = ((end.tv_sec
- start.tv_sec)
* 1000000u
+ end.tv_usec
- start.tv_usec)/1.e6;
```

calc_pi.cpp

optimized original code (single loop).

```
struct timeval start, end;
float delta;
gettimeofday(&start, NULL);

double x = 0.0;
double y = 0.0;

double total = 0.0;
double counter = 0.0;
double pi = 0.0;

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

```
    * points[i][0];
    y = points[i][1]
    * points[i][1];
    total = x + y;
    if (total <= 1) {
      counter = counter + 1;
    }
    total = 0.0;
  }
```

```

  counter = 4.0 * counter;
  pi = counter / (double) n;

  gettimeofday(&end, NULL);
  delta = ((end.tv_sec -
  start.tv_sec) * 1000000u
  + end.tv_usec
  - start.tv_usec)/1.e6;
```