Count the Integer Coordinate Points Inside the N-Sphere

Yonggang Li (18043158)

**Abstract**

This report is to introduce the algorithm and program design for counting the number of integer coordinate point inside the N dimension sphere. The performance of sequential and GPU enabled parallelized program are compared.

1. **GPU card**

There is a GeForce GTX 470 GPU card is installed on the workstation in Computer Vision Laboratory of Massey University. Here are some outputs from running device query utility from CUDA sample bin/x86\_64/linux/release/deviceQuery

Device 0: "GeForce GTX 470"

CUDA Driver Version / Runtime Version 7.5 / 7.5

CUDA Capability Major/Minor version number: 2.0

Total amount of global memory: 1279 MBytes (1341325312 bytes)

(14) Multiprocessors, ( 32) CUDA Cores/MP: 448 CUDA Cores

Total amount of constant memory: 65536 bytes

Total amount of shared memory per block: 49152 bytes

Warp size: 32

Maximum number of threads per multiprocessor: 1536

Maximum number of threads per block: 1024

Max dimension size of a thread block (x,y,z): (1024, 1024, 64)

Max dimension size of a grid size (x,y,z): (65535, 65535, 65535)

…

The device information shows that there can be maximum 1024\*65536^3 threads but there are only 448 CUDA cores, this means at runtime, there will be 448 threads at most are running, others are waiting for schedule.

1. **Algorithm**
   1. **Basic idea**

Imagine there is a smallest bounding n-dimension cube that fully encloses the n-dimension sphere. The closest integer of the edge length is 2\*floor(radius)+1. Within this cube, there are (2\*floor(radius)+1)^n integer points. If the distance of a point to the center of the n-sphere is less than radius, we can say the point is inside the n-sphere. It will be a simple mathematic problem if we have the n coordinates of the point.

The problem now is to convert a decimal integer number, num, into a number in base B. this can be solved by the following algorithm.

long idx = 0;

while (num != 0) {

long rem = num % base;

num = num / base;

index[idx] = rem;

++idx;

}

* 1. **High Dimension Issues**

When the dimension and the radius of the sphere are large, the data points need to be tested become very large, for example, in a 9 dimension space and the radius is 6, the number will be (2\*6+1)^9. The large number will cause two issues.

The first one is that it is not feasible to create a thread to test each number and store the result in a list. A limit max\_threads is set, let’s assume the total points is N, the thread number is N if N is not greater than max\_threads, otherwise it is max\_threads. Each thread, thread identifier is id, need to test the a sequence of numbers,

id, id + nthread\*1, id + nthread\*2…, until the number is greater than ntotal.

The second issue is that it takes long time for CUDA kernel to complete the task, in this case, CUDA will throw timeout exception and terminate the kernel. To solve this problem, a tuning parameter MAX\_POINTS\_PER\_THREAD is defined and there will be loop to invoke kernel and each time just process **nthreads** x **MAX\_POINTS\_PER\_THREAD** pass in a range of numbers in each call.

* 1. **Maximum Thread Number**

Threads are not free, the cost comes from two aspects

1. Creation of the thread
2. Context switching, as there are only 448 (CUDA cores) threads concurrently running, the other threads are waiting for the running thread to terminate or suspend.

As the testing of the points is computing intensive task, create too many threads won’t bring much benefits. I did a test by keep the threads per block fixed at 1024 and change the blocks per grid, we can see there is no significant performance difference between 512, 1024 and 65536. Therefore, 1024 is used in this implementation, so there will be maximum 1024x1024 threads.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Radius** | **Dimension** | **Blocks per grid (Fixed TPB 1024)** | | | |
| **10** | **512** | **1024** | **65536** |
| 4.18761 | 6 | 1.697 | 1.531 | 1.637 | 1.531 |
| 2.55379 | 7 | 0.507 | 0.47 | 0.767 | 0.471 |
| 5.1605 | 7 | 61.534 | 44.753 | 44.954 | 51.962 |
| 2.11905 | 6 | 0.303 | 0.314 | 0.31 | 0.309 |
| 7.25589 | 5 | 2.558 | 2.18 | 2.324 | 2.315 |
| 7.52157 | 7 | 537.822 | 388.908 | 389.048 | 412.969 |
| 6.86258 | 7 | 198.599 | 143.434 | 143.485 | 174.801 |
| 7.31789 | 4 | 0.532 | 0.617 | 0.586 | 0.569 |
| 2.46065 | 6 | 0.307 | 0.313 | 0.312 | 0.316 |
| 7.36344 | 4 | 0.48 | 0.472 | 0.476 | 0.473 |
| 2.6055 | 4 | 0.257 | 0.271 | 0.262 | 0.259 |

Table 1 Time for different BPGs

* 1. **Store Result**

To store the number of integer points, there are two options. One is use a global **long** **total\_count** and each thread uses **atomicAdd(&total\_count, 1)** to update it safely. Another option is use an array **long**\* counters = new **long**[nthreads], elements are all initialized with 0, is to store the testing result. As each thread has a fixed position to store how many points it tested that are inside the n-sphere. As there are maximum 1024x1024 threads, and the points number will be huge for high dimension sphere, I choose an array to void contention between threads.

1. **Program Design**

**Testing result**

Parallelized and sequential programs find the same number of integer points for any given radius and dimension. Table 1 shows the time elapsed for count different size of n-sphere.

Table 1 Time for count different n-sphere

**Conclusion**

The test demonstrates Gustafson’s Law that the more computing power the bigger problem we can solve without affecting the performance. On the other hand, the overhead cost must be considered in large distributed computing environment. This prevents our testing data fitting the formula well.