The 100 Prisoners Problem: Parallel Execution Using Graphics Processing Unit

FatemehNazemi Jenabi[1], Hamid-RezaHamidi[2]
[1]*Incubator Center, Imam Khomeini International University, Qazvin, Iran*
[2]*Computer Engineering Department, Faculty of Engineering and Technology, Imam Khomeini International University, Qazvin,Iran*
nazemijenabi@org.ikiu.ac.ir

***Abstract*— An existing optimal strategy to solve the 100 prisoners problem is to assume that its success probability is independent of the number of prisoners. However, the execution time depends on the size of the problem. For this strategy, both sequential and parallel implementations are applicable. In this paper, we compared the execution times of the sequential and parallel algorithms to see how they vary when the problem size increases.**

This paper posits that in spite of the parallel nature of this strategy, it will not fully benefit from the GPU implementation. The results show that in spite of the GPU's high memory latency overhead, the parallel implementation will outperform the sequential of larger problem sizes. For the problem size of 100, the GPU implementation using global memory yields a speedup of 0.012. The achieved speedup reaches 1.652, as the problem size increases to 100,000. For the problem size of 100, the implementation using GPU's shared memory runs 8 times faster than the one using global memory.

***Index Terms*—100 prisoners problem; Graphics Processing Unit ; Pointer-following strategy.**

I. INTRODUCTION

The 100 prisoners problem is a mathematical problem in probability theory and combinatorics. Its original version was first posed by Peter Bro Miltersen [1]. The pointer-following strategy is an optimal solution, which provides a 30% success probability [2]. It is proved that the success probability of this strategy is independent of the number of prisoners [3]. Indeed, this independence does not apply to the execution time with O (n^2) time complexity.

The structure of the pointer-following strategy suggests possible improvements through parallelization. In this paper, we present a parallel execution on GPU. First, the 100 prisoners problem and its optimal strategy are explained. Next, the sequential and parallel implementations (on CPU and GPU respectively) are explained. Finally, the achieved speedups are compared and analyzed.

II. THE 100 PRISONERS PROBLEM

In the 100 prisoners problem, the prisoners are given a chance to survive. The prisoners enter a room containing 100 boxes, one by one. Each box holds a name. As each of prisoners enter the room, they are allowed to open half of the boxes, one after another. If all the prisoners find their own names, all of them will be spared. Before the game starts, the prisoners can discuss and agree on a strategy. Once the first prisoner enters the room, no communication is allowed [3].

If every prisoner selects 50 boxes at random, the probability that a single prisoner finds his name is 50%. Therefore, the probability that all prisoners find their names is the product of the single probabilities, which is $(1/2)^{100}$, a vanishingly small number. The situation appears hopeless.

Surprisingly, pointer-following is a strategy that provides a survival probability greater than 30%. The key to success is that the prisoners do not have to decide beforehand which boxes to open. Each prisoner can use the information gained from the contents of the previously opened boxes to decide the next drawer to be opened. Another important observation is that the success probability of one prisoner is not independent of the success of the other prisoners. In fact, this is the optimal strategy [2]:

To describe the strategy, not only the prisoners, but also the boxes are numbered from 1 to 100, for example row by row, starting from the top left drawer. The strategy is as follows [3]:

1. Each prisoner first opens the box with his own number.
2. If this box contains his number, he is done and is considered successful.
3. Otherwise, the box contains the number of another prisoner, and he opens the box with this number.
4. The prisoner repeats steps 2 and 3 until he finds his own number or has opened 50 boxes.

This approach ensures that every time a prisoner opens a box, he either finds his own number or the number of another prisoner he has not yet encountered.

III. IMPLEMENTATION OF THE STRATEGY

Unlike the success probability, the execution time is not independent of the number of prisoners. We are going to study how the execution time varies if the problem size is increased.

We could deploy either a sequential or a parallel view for implementation and still adhere to the conditions of the problem. We choose GPU as our parallel platform. In order to utilize GPU's high processing power, recognizing the properties and features of successful applications on GPU is required.

A. The Properties of Successful Applications on GPU

GPUs were first designed as configurable graphics processors meant to deal with the real-time processing

requirements of computer games. With an increasing number of tools and libraries, which are introduced into the market, GPUs are now widely used for general purpose applications [4]. A comparison of the features of the two processors is presented in Table 1.

Table 1
CPU vs. GPU [5]

CPU	GPU
Multiple full cores	Hundreds of cores
Different kinds of memory to reduce latency	Fast access to limited on-die memory
Reduces latency via powerful caches	Hides latency via calculations
Optimized for high performance sequential codes (provides cache or branching prediction)	Optimized for a high volume of arithmetic computation with inherent parallelism (floating-point operations)
Provides per thread performance	Focuses on throughput
Great for task parallelism	Great for data parallelism
Has proper branching ability	Faces performance reduction when encounters divergence

With these features in mind, we cannot expect the pointer-following algorithm to benefit optimally from GPU architecture. It has a low arithmetic intensity, in spite of being highly parallel in nature. In addition, it also has many irregular memory accesses. As a result, its implementation on a GPU will suffer from a significant overhead.

B. Parallel implementation on GPU

Both the sequential and parallel implementations are based on vectors. Each cell of the vector represents a box. The indices indicate the box numbers and the values indicate the box contents. At the beginning, the vector is filled with sequential integers from *0* to *(n-1)* where *n* is the number of prisoners. Then, using the Fisher-Yates shuffle algorithm [6], a random permutation is generated. This vector models the boxes and it is all done by CPU on the host memory. For the parallel implementation, the vectors are then moved to the device memory of GPU.

In the sequential version, the prisoners will enter the room one after another. This algorithm contains two nested for loops leading to an $O(n^2)$ time complexity.

The pseudocode for the CPU sequential version is:

```
StartTimer(&t1 );
for (inti=0; i<n; i++)
        for (int j=1; j<n/2; j++)
                if (Results[i] !=i)
Results[i]=Boxes[Results[i]];
printf( "Time = %.3f\n", StopTimer( t1 ));
```

In the parallel version, we can suppose 100 identical rooms. Since each prisoner's attempt to find its own name is independent of other prisoners, they can follow the strategy concurrently. In the context of CUDA[1], this means *n* CUDA threads. In the simplest form, a CUDA kernel is executed *n* times by *n* different CUDA threads.

Thrust library [7] is used to implement the parallel version of the strategy. High-level libraries such as Thrust, can avoid programmers from the low-level complexities, like memory management. However, the results may not be as efficient as a fully optimized application written in CUDA. To test this, we compared the results of a Thrust-based implementation to a not fully optimized CUDA kernel. We did this comparison for small problem sizes, where using shared memory is possible. Shared memory is similar to a user-managed cache: fast but limited. Therefore, we can benefit from it for smaller problem sizes by doing a user-defined kernel. For bigger problem sizes where shared memory's capacity does not suffice, we have to use the slow global memory.

The pseudocode for user-defined parallel version is:

```
Timer Starts
__global__ void OpenBoxes(int* A)
{
        __shared__ int Result[n], Initial0[n];
        inti = threadIdx.x;
        for (int j=0; j< ((n/2)-1); j++)
                if(Result[i]!=i)
                        Result[i] = Initial[Result[i]];
}
Timer Stops
Elapsed time is calculated and printed
```

The Pseudocode for the Thrust-based parallel version is:

```
Timer starts
for (inti=0; i<n/2; i++)
{
        thrust::copy(
            thrust::make_permutation_iterator (
        BoxContent.begin(),
        risonerNum.begin()),
            thrust::make_permutation_iterator (
        PrisonerNum.begin(),
        PrisonerNum.end()),
       PrisonerNum.begin());
       thrust::transform_if(
            thrust::make_zip_iterator(
                    thrust::make_tuple(
        sequence.begin(),
        PrisonerNum.begin())),
           thrust::make_zip_iterator(
                    thrust::make_tuple(
       sequence.end(),
       PrisonerNum.end())),
       BoxContent.begin(),
       change_my_nodes,if_condition());
}
Timer Stops
Elapsed time is calculated and printed
```

[1]Compute Unified Device Architecture: NVIDIA's parallel processing platform.

Thrust has proven its ability to be efficient for data intensive applications. Kaczmarski and Rzążewski have used permutation generation as their test application. This is because they believed it has four considerable features, including a high number of read/write operations. This is the most important feature that needs to be considered, when implementing 100 prisoners problem. The permutation generation also uses constant iterator, counting iterator and transformation iterator. We used transformation iterator for the implementation of 100 prisoners. Their evaluations showed that when implemented on Fermi architecture, Thrust can produce efficient results as good as a low level CUDA code [8].

C. Analysis of the execution times

The implementations are performed on a personal computer with these specifications: *GeForce GT 425M (Fermi Architecture, Compute Capability 2.1), Intel Core i7 (1.73 GHz), Windows7, CUDA5, Visual Studio 2012*.

The average execution time of the strategy is measured for different problem sizes. The codes are executed 5 times and the average of these 5 execution times is reported. The results are presented in Table 2 and Table 3.

In order to get realistic results for the GPU speedup, memory transfer times need to be either included or proven to be of no concern. Gregg and Hazelwood stated that "the location of the data that is being processed before, during, and after kernel execution" is an underestimated factor [9]. So, the GPU processing time without memory transfers before and after, may not be a proper measurement of actual GPU performance. As described in section B, we have an initial data transfer. We studied the implementation of the strategy only, not inclusive the whole modeling procedure of the 100 prisoners problem. Therefore, the vector generation and data preparation is beyond the scope of our study. However, our experiment on sample problem sizes showed that the calculated speedups with and without the initial memory transfer remains almost similar.

The speedup presented in Table 2 is the ratio of the sequential implementation to the Thrust-based parallel. The speedup presented in Table 3 is the ratio of the sequential implementation to the user-defined parallel for small problem sizes. The growth of the speedup with increasing problem sizes is shown in Figure 1 and Figure 2.

Table 2
CPU vs. GPU average execution times (ms) using global memory

Problem size	Sequential execution on CPU	Parallel execution on GPU- Thrust library	Speedup
100	0.008	0.62	0.012
500	0.204	3.791	0.053
1,000	0.99	8.91	0.111
5,000	18.979	38.976	0.486
10,000	99	144	0.687
50,000	4,009	4,804	0.834
100,000	37,632	22,773	1.652

Table 3
Average execution times for small problem sizes (ms) using shared memory

Problem size	Sequential	Parallel Thrust	Parallel user-defined	Speedup
64	0.005	0.354	0.078	0.064
128	0.015	0.844	0.111	0.135
256	0.062	1.652	0.094	0.659
512	0.255	4.082	0.110	2.318
1024	1.027	7.147	0.211	4.867

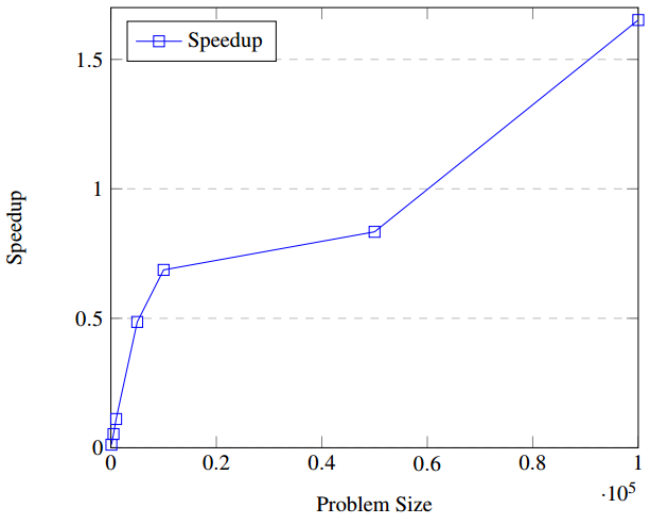

Figure 1- The GPU speedup using global memory.

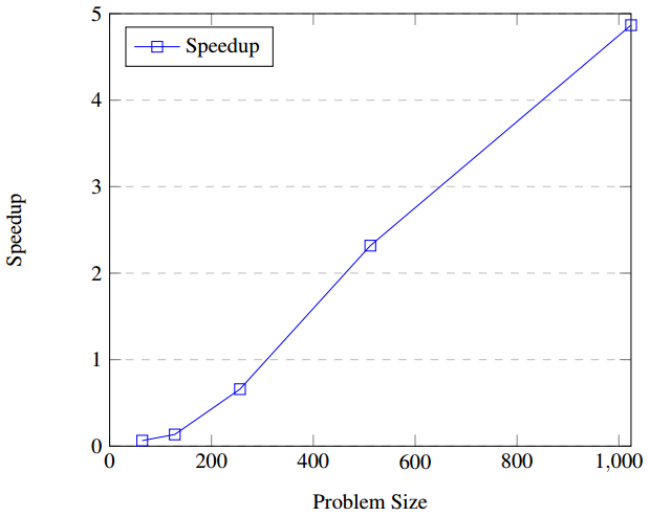

Figure 2- The GPU speedup using shared memory.

The guided analysis of NVIDIA Visual does not support unified memory profiling for devices that have computation capability less than 3.0; hence, it does not support ours, which is 2.1. Based on the examination of GPU Usage, the profiler suggestions are shown in Figure 3 and Figure 4.

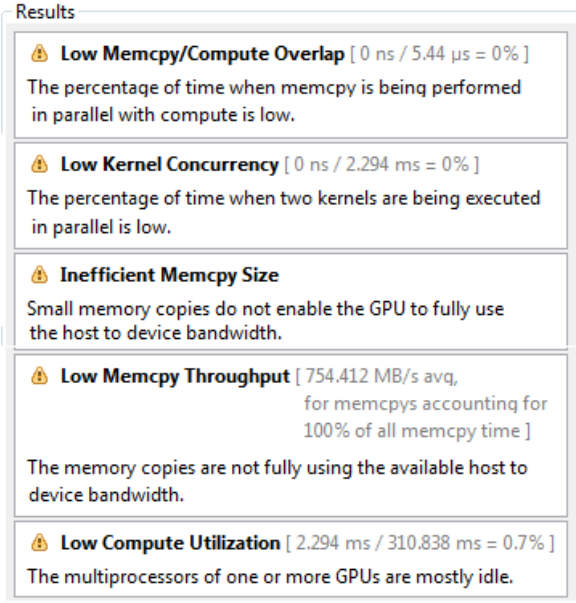

Figure 3- Analysis Results for the Thrust-based parallel implementation, Problem size=1024

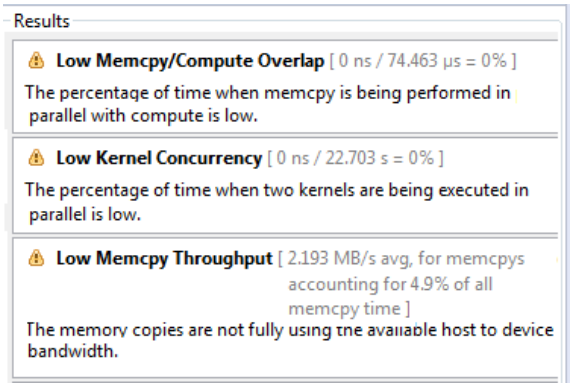

Figure 4- Analysis Results for the Thrust-based parallel implementation, Problem size=100000

The GPU's high throughput is achieved by hiding memory access latency with computation rather than high-speed caches. GPU has a high floating-point capacity and is suitable for data-parallel computations. Data-parallelism is the execution of a single program, with a high arithmetic intensity, on numerous data elements in parallel. Arithmetic intensity is defined as the ratio of arithmetic operations to memory operations [10]. GPU is limited in cache memory. Further, although the global memory[2] is big, it is slow. Therefore, it has a very high peak bandwidth on its on-board memory, which is of limited size. However, in more recent architectures, the burden of this limitation is eased to some extent [11].

The main general rules [12] for creating high performance GPGPU programs are: (1) keep the data on the GPGPU, (2) give GPGPU enough work to do, and (3) consider data reuse within the GPGPU in order to avoid memory bandwidth limitations.

Based on these specific features of the GPU, performance-friendly factors include maximization of arithmetic operations, maximization of the number of simultaneously running threads, high ratio of computation to memory access, regular memory accesses, getting and keeping data on GPU and focusing on data reuse.

However, the usual optimization techniques used to ease the burden of memory latency such as asynchronous transfer, overlapping computation with communication, minimization ofhost/device data transfer, could not be used. This is because, the implemented strategy is a pointer-following algorithm with an irregular data access pattern and a very low arithmetic intensity. For a small problem size, using shared memory is a beneficial optimization technique, as shown in Table 3. However, optimal utilization of shared memory requires the minimization of bank conflicts since shared memory is composed of memory banks that can be accessed simultaneously. However, this condition is not possible with the algorithm's irregular data access pattern, this is not possible. Further, a bigger problem size yields better resource utilization. With reference to the comparison between Figure 3 and Figure 4, the increased problem size results in improved GPU usage for specific factors. These factors include Memcpy size and compute utilization.

As presented in Table 2, speed up increases by increasing the problem size. This relationship is not the results of the process of a larger data to be processed and we are using the GPU efficiently. Rather, this is because the $O(n^2)$ sequential algorithm's execution time grows faster than the parallel algorithm's. As we can see in Figure 5, the growth of CPU's execution time is faster that GPU's. The horizontal axis shows the problem size growth. The vertical axis shows the execution time growth caused by the problem of size growth.

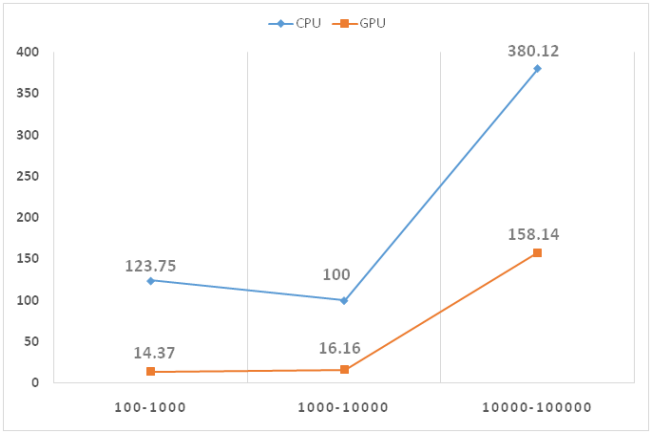

Figure 5 -Execution Time Growth: CPU vs. GPU

IV. Conclusion

Although the independent nature of each prisoner's search procedure indicates a high potential for parallelism, it is not straightforward in practice. A high volume of read/writes from/to the slow global memory is GPU's bottleneck. GPU is designed to hide latency through intensive computations, a property, which is entirely absent in this algorithm. In spite of the memory overhead, GPU outperforms CPU for larger problem sizes.

[2]Global memory has a high latency of up to 800 clock cycles

The results are highly dependent on the usage of graphics card since the crucial factors like global memory latency, shared memory size, or the number of shared memory banks are dictated by the hardware.

REFERENCES

[1] A. Gal, P. B. Miltersen. "The Cell Probe Complexity of Succinct Data Structures", *Proc. of 30th InternationalColloquium on Automata, Languages and Programming (ICALP'03)*, Eindhoven, 2003, pp. 332-344.

[2] E. Curtin, M.Warshauer. "The locker puzzle", *Math Intell*, vok. 28, pp. 28-31, 2006.

[3] P. P. Stanley,*Algebraic Combinatorics: Walks, Trees, Tableaux, and More*, Springer-Verlag: New York, 2013, pp. 187-189.

[4] J. Nickolls, W. Dally, "The GPU Computing Era",*IEEE Micro*,vol. 30, pp. 56-69, April 2010.

[5] F. Nazemi Jenabi, H. Hamidi, *Parallel computing on Hybrid CPU-GPU systems*, Payam Noor University, Tehran, Iran, 2014, Unpublished Master's thesis.

[6] D. E. Knuth, "Seminumerical algorithms", in *The Art of Computer Programming*, 3rd ed. vol. 2, Boston: Addison–Wesley, 1998, pp. 145–146.

[7] Thrust Quick Start Guide v8.0, NVIDIA Corporation, 2017.

[8] K. Kaczmarski, P. Rządewski,"Thrust and CUDA in Data Intensive Algorithms", in *New Trends in Databases and Information Systems*, M. Pechenizkiy and M. Wojciechowski, Ed. Addison Springer-Verlag Berlin Heidelberg, 2013, pp. 37-46.

[9] C. Gregg, K. Hazelwood. "Where is the data? Why you cannot debate CPU vs. GPU performance without the answer", in*Proc. of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '11)*, Washington, DC, USA, 2011, pp. 134-144

[10] J. D. Owens, M. Houston, D.Luebke, S. Green, J. E. Stone, J. C. Phillips. "GPU Computing", in *Proc. IEEE*, vol. 96, no. 5, pp. 879-899, May 2008.

[11] Kihgariff E, Fernando R." The GeForce 6 Series GPU Architecture", in *GPU Gems2: programming techniques for high- performance graphics and general-purpose computation*, M. Pharr, Ed. Addison-Wesley Professional, 2005, pp. 471-492.

[12] R. Farber,*Cuda Application Design and Development*. Waltham USA : Morgan Kaufmann, 2011, pp. 13-15.